IN-61
207528
62 P

NASA Contractor Report 194475

Design and Implementation of a Distributed Version of the NASA Engine Performance Program

Jeffrey T. Cours
Ohio State University
Columbus, Ohio

March 1994

Prepared for
Lewis Research Center
Under Grant NAG3-1369

National Aeronautics and
Space Administration

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to Distributed NEPP

The NASA Engine Performance Program, or NEPP, is a program NASA engineers use to design and profile aircraft engines [9]. The user will typically specify the engine's configuration, desired performance, and constraints; NEPP will then calculate the sizes of the engine components necessary to satisfy all the design parameters. Once it has sized the engine components, the program goes on to calculate how the engine will perform in *off-design cases*: altitudes, speeds, and throttle settings different from those at which it designed the engine. These off-design cases provide a natural point to break NEPP into parallel execution streams.

There are a number of practical reasons for speeding up NEPP by running it in parallel. The total time NEPP takes to design and profile the engine depends very strongly on engine complexity, which profiling options the engineer uses, and how many off-design cases she chooses to run: test cases on an IBM RS/6000 model 560 have ranged in time from 5 minutes to over 35. In fact, the only real limitation on the program's run time is the fact that the engineer naturally tends to limit the complexity of the engine and parameters, and the number of results the program produces, to keep the maximum run time within the limits of patience: given a faster version of the program, she could easily increase the complexity of the problem to use the additional computational power. Furthermore, [8] describes a new program called IPAS which uses the NEPP code, running it hundreds or thousands of times in the course of designing and optimizing an entire airframe and engine combination. If each NEPP run takes 5 minutes, and IPAS calls NEPP 100 times, IPAS will spend over 8 hours just using NEPP, presenting an added incentive for developing a parallel, fault-tolerant version of NEPP.

A less pragmatic but equally pressing reason for converting NEPP to run in parallel is that the conversion presents the chance to develop a distributed application that operates under some unusual constraints. Chapter 2 describes several proposals from the literature for ways to develop a distributed program. However, most of these proposals work best for a transaction processing-type application in which assignments arrive at different processing nodes throughout the system and at different times: many of these approaches consider having all the assignments arriving at the same processor at the same time to be a worst-case situation.

In contrast, NEPP draws its work from a set of input files the engineer creates beforehand, allowing it to control how many tasks arrive at a given time. Furthermore, there is only one copy of the input files available at the beginning of the NEPP run, so all the tasks inherently arrive at the same node. NEPP's design constraints allow, and even require, a different approach from the ones chapter 2 presents; chapter 3 describes the algorithms the program uses.

The new version of NEPP must meet requirements in addition to efficiently running in parallel. Engineers will use the new program as a design tool, and other code like IPAS will rely on it. Therefore, the new version of NEPP has to behave like its sequential counterpart: it must have the same user interface, return the same results, and be at least as resistant to crashing as Sequential NEPP. *Distributed NEPP*, the new, parallel version of NEPP, is a fault-tolerant, parallel application that looks like its sequential counterpart to the end user.

Distributed NEPP runs on a cluster of workstations connected by a network. The program uses a standard communications package to coordinate the activities of processes on each of the workstations, allowing it a great deal of flexibility in the types of networks it can use. It makes very few assumptions about the underlying file system and is capable of connecting a geographically separate, heterogeneous collection computers to work in parallel and solve the computational problem. Furthermore, it is designed to share the workstations with other applications, and its dynamic load-balancing system allows it to quickly adapt to changing system loads.

Distributed NEPP's other features include the fact that, given a sufficiently complex program to analyze, the program is faster than the original, sequential version of NEPP. It keeps the same user interface and input file format as Sequential NEPP so it will be easy for the aviation engineers who use NEPP to switch to the new application. Also, to give Distributed NEPP the same capabilities as sequential NEPP, and to avoid having to maintain two versions of the same program, Distributed NEPP uses the same library of Fortran subroutines that sequential NEPP draws upon. Finally, the program is fault-tolerant: if a processor crashes, or becomes so heavily-loaded that Distributed NEPP can no longer use it to do useful work, Distributed NEPP will recover and continue processing using the remaining nodes. As this report will discuss later, due to the limitations of the communications package the program uses, if the communications software on a processing node actually crashes then Distributed NEPP will crash as well, but it will recover gracefully if one or more nodes begins to run so slowly that it effectively crashes. Furthermore, a planned upgrade to the newest version of the communications package will remove the restriction against a node actually crashing, and, using the new package, it should be relatively easy to modify Distributed NEPP to detect the fact that a node has recovered and to begin to use that node to do useful work again.

1.2 PVM Communications Package

To reduce Distributed NEPP's complexity and to keep it from becoming dependent on any one particular network protocol, this project uses the PVM software package [5]. PVM acts as an interface between Unix applications and lower-level network protocols so that, to the programmer, a networked collection of Unix workstations looks like a partitioned-memory multiprocessor with extremely high process initiation and message set-up times. PVM provides facilities for remote process initiation and termination, data format conversion, and one-to-one and broadcast communication. It can link a heterogeneous collection of machines and network links into a single virtual multiprocessor: version 2.4, the version Distributed NEPP uses, supports computers ranging from the Sun SPARC to the Cray Y/MP.

Distributed NEPP gains several benefits from using PVM to handle all its inter-process communication. The first advantage is that the program avoids becoming restricted to a particular type of computer or network protocol. Also, PVM's abilities to link heterogeneous computers allows Distributed NEPP users to run the application on whatever computers they have on hand. Finally, because PVM is an evolving software package, as PVM's capabilities grow, so will Distributed NEPP's.

Chapter 2

Background

2.1 Survey of Fault Tolerant and Load Balancing Schemes

The literature describes several possible approaches to distributing a set of jobs to a networked collection of processors. Some of these approaches address load balancing, others focus on fault-tolerance, but for one reason or another, most are unsuitable for the Distributed NEPP application, necessitating the development of a new approach.

2.1.1 Arrival-, End-, and Continually-Balance Scheduling

Ben Blake surveys several scheduling and load balancing systems in [2]. Three of the more interesting include arrival-balanced scheduling, end-balanced scheduling, and continually-balanced scheduling. An *arrival-balanced scheduler* places each incoming job on the processor that will finish it first, given the current system state, but it does not include provisions for moving jobs from one processor to another after it has already scheduled them. In contrast, an *end-balanced scheduler* places each incoming job on the processor where it initially arrives. Idle processors in the system query the other processors and take jobs from the most heavily-loaded one. A *continually-balanced scheduler* generates a near-optimum schedule on a single processor, based on *a priori* knowledge of how long it will take to complete each job. The scheduling processor then propagates the information to the other nodes in the system. In this scheme, incoming jobs go to the processors that should finish them first, but the scheduler is free to migrate jobs as necessary.

Blake discusses three measures of scheduler performance: the total time it takes for the system to run the application excluding scheduler overhead, the number of times a given scheduler executes during the application run, and the number of job migrations the scheduler triggers. His simulation results assume that the job execution times have a uniform distribution and that, in general, jobs appear on processors throughout the system rather than all arriving at a particular processor. Furthermore, while [2] does not explicitly state the assumption, a close reading of the paper seems to indicate that all processors in the system ran at identical, and constant, speeds. (This assumption, in particular, does not apply to the Distributed NEPP system, which has to be able to operate in a heterogeneous, dynamic environment.)

Blake's simulations show that the complexity of a scheduler can have a significant impact on its performance. In particular, the more complex continually-balanced scheduler outperforms the other two when several jobs arrive at a given processor simultaneously, with long periods of no job arrivals in between. On the other hand, the simpler end-balanced scheduler outperforms its more complex counterpart when the number of jobs which arrive simultaneously is small, but the jobs arrive frequently. The difference in performance comes from the additional overhead that continually-balanced scheduling incurs. Depending on the job arrival pattern, the end-balanced scheduler will execute from 0 to $J-1$ times, where J is the number of jobs the application must complete to finish execution. Also, the only global information the end-balanced scheduler requires is the current load on each processor. On the other hand, the continually-balanced scheduler executes once for each set of jobs that arrives simultaneously, which explains its superior performance

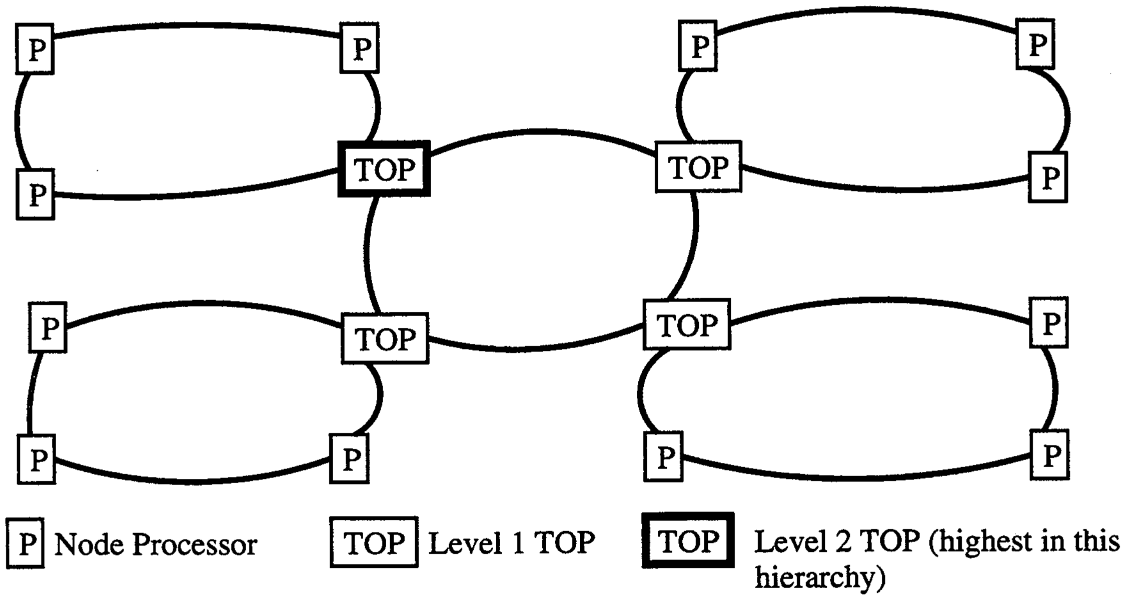

Figure 2.1: Multiple Virtual Rings—overall organization

when the job arrival pattern shows a few large lumps of arriving jobs rather than frequent arrivals of a small number of jobs. The continually-balanced scheduler also requires more global information: to perform its calculations, it has to know how long each job on each processor will take to complete. Finally, Blake's simulations indicate that the continually-balanced scheduler tends to trigger more job transfers than the end- or arrival-balanced schedulers, adding to the overhead it requires.

Unfortunately, Blake's results do not apply very well to Distributed NEPP. Blake does not address the problem of fault-tolerance: if a processor receives a job and then becomes faulty, some other processor must do the work of the faulty one for the application to finish. Also, in Distributed NEPP, all the jobs arrive at the same processor simultaneously, but the continually-balanced scheduler, the one best suited for such a situation, requires *a priori* knowledge of when each job will finish, something that is nearly impossible to determine in the dynamic environment in which Distributed NEPP operates. However, the Simple Distribution algorithm that chapter 3 describes is, in many respects, similar to Blake's arrival balanced scheduler, in which arriving jobs go to the processor most likely to finish them earliest.

2.1.2 Multiple Virtual Rings

Another approach to dynamic load balancing is the *Multiple Virtual Rings*, or *MVR*, protocol J. G. Vaughan presents in [15] which organizes the processors into a hierarchy of rings (figure 2.1). The Multiple Virtual Rings protocol calls for tokens to circulate around the rings at each hierarchical level, determining and balancing loads. Vaughan calls the processors that manage the tokens *Token Origination Processors* or *TOPs*. At the lower level rings, the TOP for the ring is the processor that connects the ring to the next higher ring in the hierarchy; the algorithm arbitrarily designates one of the processors in the highest level ring as the ring's TOP. At the start of each load balancing cycle, the TOP in each ring emits a token. The token circulates around the ring and determines the maximally- and minimally-loaded ring processors. When the token returns to the TOP, the TOP sends a message through the maximally loaded-processor to the minimally-loaded one that gives the minimally-loaded processor the option of accepting work from the maximally-loaded one. TOPs are also responsible for maintaining information about the load level of their rings and transferring work up and down the ring hierarchy, so that the load across the entire system will eventually balance.

Vaughan's simulation results indicate that MVR is effective at balancing the system load provided that, before the algorithm starts, the jobs are spread out through the processors. However, [15] indicates that if one processor has all the jobs at the beginning of the load-balancing process, the most common situation for Distributed NEPP, the system takes a significant amount of time to reach a balanced state.

MVR's main advantage is that its distributed approach to load balancing helps to prevent bottlenecks, since processors need load information from others in a local neighborhood rather than globally. The drawbacks to the scheme are its complexity, the fact that it requires the ability to checkpoint and transfer jobs, and the relatively high number of messages per job transfer. In fact, as [15] indicates, the number of overhead messages divided by the number of jobs the processors transferred ranged from 15 to 35, implying that overhead could be quite expensive on a system with high communication set-up costs. Furthermore, [15] does not address the question of what to do if a processor fails.

2.1.3 Reliability Optimization

Vaughan and Blake's approaches ([15] and [2]) try to minimize the amount of time the system takes to run an application, but they do not consider system reliability or fault tolerance. In [12], on the other hand, Sol M. Shatz, Jia-Ping Wang, and Masanori Goto attempt to maximize the chance that the system will finish the job, but they do so at the expense of poor load balancing. Their approach increases reliability not through adding redundant hardware or software but by allocating jobs to processors in such a way as to minimize the chance that a processor or link will fail while the application is running. Using *a priori* knowledge of each processor's and link's reliability and the amount of time each job in the application takes to run, they define a cost function which increases as reliability decreases. In essence, the cost function captures the idea that shorter jobs should go to less reliable processors to minimize the chance that the processor will fail during the job, and also that if jobs must communicate with each other, then those jobs requiring the most communication should go to the processors with the most reliable communications links.

Once they have defined their cost function, Shatz et. al. attempt to allocate jobs to the processors in such a way as to minimize the cost and thereby maximize reliability. They present an algorithm, based on the Artificial Intelligence algorithm "A*", which will find the optimum allocation of jobs for maximum reliability. Unfortunately, in the worst case the exact algorithm requires time that is exponential in the number of jobs, so Shatz et. al. also present a number of polynomial-time algorithms that will find near-optimal solutions based on a set of heuristics:

1. More reliable processors should have more work to do, and therefore spend more time calculating, than less reliable processors.
2. Jobs that communicate heavily with each other should go on the same or nearby processors.
3. Jobs that communicate most heavily with several other jobs should go on "centrally-located" processors, processors which can most reliably reach other processors through communication links.

Using these heuristics, they develop algorithms that run in times ranging from $O(J^2)$ to $O(pJ)$, where p is the number of processors in the system and J is the number of jobs the application must schedule.

In [12], Shatz et. al. do not analyze the effects their approach has on total execution time. However, note that heuristic 1, in particular, tends to work against load balancing by exaggerating the load on more reliable processors at the expense of less reliable ones. Furthermore, they do not address the question of how to calculate or estimate processor or communication link reliability in a dynamic, heterogeneous system such as the environment Distributed NEPP supports. Accordingly, as later sections will explain, Distributed NEPP attempts a more dynamic approach to task allocation and incorporates redundancy in time, resulting in a program that uses computer resources less efficiently than the scheme [12] proposes, but which achieves better load balancing along with high reliability. On the other hand, future versions of Distributed NEPP could benefit from incorporating heuristics 1 through 3, above, into the scheduling process, assuming the program could somehow estimate system reliability values and job completion times.

2.1.4 Contract Net Protocol

None of the scheduling approaches so far address both load balancing and fault-tolerance. One approach that does appears in [13], where Reid G. Smith presents the *Contract Net Protocol.* In the Contract Net

Protocol, processors with an excess of jobs, or with jobs that would be better suited to other processors, broadcast a message describing the job they wish to *contract out*, or deliver to another processor. Processors that are idle can then respond to this job announcement message and *bid* on the *contract*. The contracting processor, the one offering the job, selects one of the bidders and assigns it the job. All messages include an expiration time. If a contractor does not receive any responses to its job announcement by the expiration time it attached to the message, it knows that there are no nodes in the system capable of handling its job. Similarly, if a bidding node does not receive a response to its bid by the expiration time, it knows that it did not get the contract. On the other hand, since a node is free to choose whether or not to bid on a job, the system inherently balances its load as long as only lightly-loaded nodes bid on job contracts. Furthermore, the contracting node can select the most suitable bidder to receive the job, providing the intriguing possibility of incorporating heuristics 1–3 from [12].

In addition to inherently balancing the system load and handling a dynamic, heterogeneous environment, the Contract Net Protocol's bidding system allows nodes to enter and leave the system very easily. A contractor can request status reports from a node running one of its jobs, so the contractor will know if the node fails and can solicit new bids for the contract. If a new node enters the system, it can immediately begin bidding on new job assignments without having to update information on other nodes. Also, the Contract Net Protocol avoids bottlenecks due to centralized control.

Unfortunately, the Contract Net Protocol has one major drawback: high communications overhead. Transferring a job or group of jobs from one node to another requires, under ideal circumstances, the following messages:

- one multicast to announce the job
- one message from each bidding node to bid on the job
- one message from contractor to bidder to award the job

While each message could be quite small, the protocol requires a large number of them for each job, or group of jobs, that one processor transfers to another. This large number of messages could introduce a great deal of inefficiency when running on any system suffering from high communications set-up costs, such as the communications package Distributed NEPP uses. The large number of messages appears to result from the fact that no processing node maintains information about other nodes' load levels, since if nodes knew each others load levels, the contractor could assign the job to the recipient directly. This same factor makes the protocol's fault-tolerant aspects simple and elegant, so changing it might prove difficult.

2.1.5 Summary

Unfortunately, none of these approaches is entirely suitable for Distributed NEPP. Vaughan's Multiple Virtual Rings and Smith's Contract Net Protocol both suffer from requiring a large number of messages per job transfer, a defect that could devastate Distributed NEPP's performance given the high message set up cost it faces. Furthermore, neither approach has been optimized for the case in which all the jobs arrive simultaneously at a single processor, which is the usual case when Distributed NEPP begins. Shatz et. al. provide a useful set of reliability-improving heuristics that later versions of Distributed NEPP might incorporate, but they do not address the question of how to obtain accurate reliability estimates for the components involved in the networks they study, and their approach increases reliability but does not guarantee that the system will be reliable in the face of node failures. Finally, the load balancing algorithms that Blake presents do not explicitly address the problem of fault-tolerance. However, as Chapter 3 shows, his arrival-balanced scheduling algorithm is similar in many ways to the algorithm Distributed NEPP uses.

Chapter 3

Algorithms

3.1 Distributed NEPP's Job Allocation Algorithm

3.1.1 Overview

The job scheduling algorithms in section 2.1 all have their strengths and weaknesses, and the algorithm Distributed NEPP uses is no exception to the trend. Since no perfect job allocation algorithm exists, one of the challenges in developing Distributed NEPP was to create an algorithm whose strengths would improve the program's performance but whose weaknesses would have very little or no impact on this specific application.

Figure 3.1 shows the organization of the input file that both the sequential and the distributed versions of NEPP read. The *design point* portion of the file tells NEPP the components that make up the engine it will analyze, how those components connect to one-another, what sort of performance the engineer expects from the engine, and for what throttle setting, altitude, and speed NEPP should calculate the sizes the various engine components. Then, each *off-design case* has NEPP calculate the engine's performance at a throttle setting, speed, or altitude different from the ones the design point specifies, but using the same engine component sizes NEPP calculated for the design point. The off-design cases provide a natural way to break sequential NEPP into threads that can execute in parallel. Each off-design case represents a separate job that Distributed NEPP can schedule on a processor. The off-design cases do not strongly depend on one-another for most engine configurations, although, as this section discusses later, having the nodes process them in the same order in which they appear in the input file generally improves execution time. Since the off-design cases do not strongly depend on each other, the algorithms this section presents treat them as independently-schedulable, atomic jobs for the node processors to compute; the more sophisticated algorithms try to schedule consecutive off-design cases on the same node processor. Finally, to keep the terminology in this discussion as general as possible, the rest of this chapter refers to the off-design cases simply as *jobs*.

In order be as flexible as possible, Distributed NEPP does not assume that the cluster of processors it is using has any sort of network file system. This restriction against assuming a network file system complicates Distributed NEPP's design slightly, but it lets users set up ad-hoc networks of machines at geographically separate sites and makes a centralized scheduling and load balancing approach more practical. Instead of

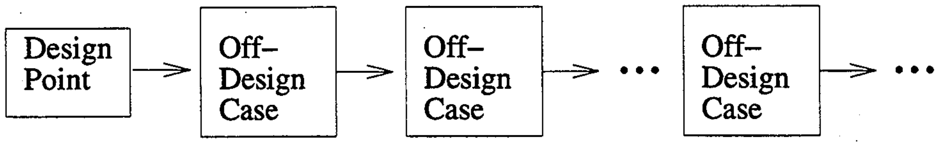

Figure 3.1: Organization of Distributed NEPP's input file.

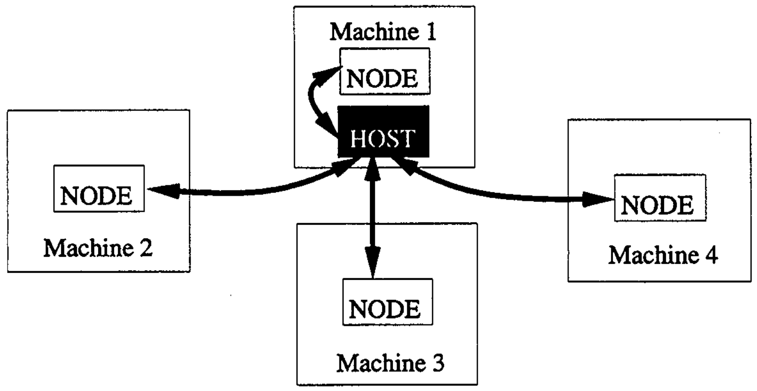

Figure 3.2: Arrangement of host and node processes on a cluster of 4 machines.

assuming that all processors can get access to the input files, as would be the case with a network file system, Distributed NEPP assumes that only one processor can access them, but that the one processor has access to all the input files. To run Distributed NEPP, the user starts a *host* program on the one processor that can access the input files. The host, in turn, starts a *node* program running on each processor, including the one where the host resides. Figure 3.2 shows a 4-machine cluster with one host and four node processes, with arrows representing communication paths between the processes. The host process has guaranteed access to the input files, so it is responsible for assigning work to the nodes, collecting their results, and writing the results to the output file. This centralized scheme simplifies load balancing and fault-tolerance at the expense of requiring the host to explicitly pass all information to and from the nodes.

3.1.2 Simple Distribution Algorithm

The simplest approach to distributing the load, and the one that the first version of Distributed NEPP used, is for the host to send individual jobs to the nodes on a first-come, first-served basis. The first step in the algorithm is for the host to read the design point information and broadcast it to the nodes. The host and the nodes compute the design point information, determining the sizes of the engine components the engineer has specified in the input files. As each node finishes its computation, it sends a READY message to the host. Having both the host and the nodes calculate the design point is not strictly necessary, but it proved to be the easiest way to initialize the data structures in the NEPP library.

As soon as the host gets a READY message from a given node, the host sends that node the next job from the input file. When each node finishes calculating the results of the job the host has sent it, the node sends the results back to the host in a DONE message. Upon receiving a DONE message, it unpacks the results, and, if there are any jobs left in the input file, the host sends the next one to the node. This process continues until there are no jobs left in the input file. For the sake of discussion, call this approach, which appears below in pseudocode form, *Simple Distribution*:

Algorithm: Simple Distribution, Host Portion

```
begin
    broadcast design point and associated information
    calculate design point
    while (there are jobs left in the input file)
    begin
        (blocking) receive the next READY or DONE message from a node
        reply to the node with the next job
    end
```

```
        broadcast a DIE message
    end
```

Algorithm: Simple Distribution, Node Portion

```
    begin
        (blocking) receive design point and associated information
        calculate design point
        send READY message to host
        (blocking) receive the next message from the host
        while (have not received a DIE message from the host)
        begin
            process the job
            send the results to the host in a DONE message
            (blocking) receive the next message from the host
        end
    end
```

The Simple Distribution algorithm has many advantages. The approach does not require the host or the nodes to maintain a job queue, the load-balancing granularity is one job, and this algorithm balances the loads very effectively. Since a node never gets a new job until it has finished with its current one, faster nodes automatically receive more work than slower ones, and the algorithm reacts quickly and naturally to changing system loads and processor speeds. Also, the algorithm's communications overhead is quite low compared with the approaches in Chapter 2: it requires only one message per job transfer if, like the algorithms in section 2.1, we neglect the messages that return the nodes' results to the host. In many ways, this algorithm is similar to Blake's arrival-balanced algorithm, which places newly-arriving jobs on the processor most likely to finish them first.

Unfortunately, the Simple Distribution algorithm has some very serious drawbacks, including the fact that it makes no provisions for fault-tolerance: if a node crashes after the host as already assigned it a job, the host will continue assigning jobs to nodes and receiving results until it reaches the end of the input file. Then, the host will hang forever waiting for the results from the crashed node.

Another problem with the algorithm is that, as this section mentioned earlier, the off-design cases are not entirely independent. The NEPP library uses an iterative algorithm to calculate the results for each off-design case, and it uses the results of the last off-design case as a starting point for converging to the next. Depending on the engine configuration, if the library starts its calculations using the wrong set of data, it may take significantly longer to converge to a result. In fact, given a particularly sensitive engine configuration, the code may not converge at all. Since the Simple Distribution algorithm distributes one job at a time, on a first-come, first-served basis, it tends to scatter jobs almost randomly among the nodes: even if the engineer has arranged the input file in such a way as to speed convergence, Distributed NEPP would not be able to use that fact to its advantage, and it could suffer performance degradation or, in the worst case, even fail to converge on input files that Sequential NEPP can process with no trouble.

The final drawback to using the Simple Distribution algorithm is the fact that, while it requires only one message per job transfer, the algorithm still has to pay the PVM communication package's high message set-up costs once for each job the host sends to the nodes. With the job compression system section 3.2 describes, a single job message generally contains about 4 double precision numbers and 4 integers. Furthermore, the size of the message decreases the closer each job's parameters are to the preceding one's. Sending consecutive jobs to the same node, and packing more jobs into each assignment message, would result in a more efficient algorithm.

3.1.3 Multiple Distribution Algorithm

The next step in complexity is the *Multiple Distribution* algorithm, an algorithm that packs multiple jobs into each job assignment message. This algorithm requires that each node have a job queue. As the nodes send back their results, the host monitors the nodes' queues, and, whenever the number of job assignments in a node's queue drops below a threshold value, the host sends that node a single assignment message containing several job assignments. Here is the pseudocode for the Multiple Distribution algorithm:

```
Algorithm: Multiple Distribution, Host Portion
begin
    broadcast design point and associated information
    calculate design point
    while (there are jobs left in the input file)
    begin
        (blocking) receive the next READY or DONE message from a node
        if node's queue size is less than minimum threshold size
        begin
            pack an assignment message with enough jobs to fill the node's queue to
              maximum threshold size
            send the assignment message to the node
        end
    end
    while (the host has not received all results)
    begin
        (blocking) receive the next READY or DONE message from a node
    end
    broadcast a DIE message
end
```

```
Algorithm: Multiple Distribution, Node Portion
begin
    (blocking) receive design point and associated information
    calculate design point
    send READY message to host
    (blocking) receive the next message from the host
    if the message was not a DIE message
    begin
        enqueue, in order, all job assignments from this message
    end
    while (have not received a DIE message from the host)
    begin
        while (there is not a message waiting from the host)
          and (the job queue is not empty)
        begin
            dequeue the next job from the job queue
            process the job
            send the results to the host in a DONE message
        end
        (blocking) receive the message from the host
        if the message was not a DIE message
        begin
            enqueue, in order, all job assignments from this message
        end
    end
end
```

While the Multiple Distribution algorithm requires the nodes to maintain job queues and is more complicated than the Simple Distribution algorithm, the former has several advantages over the latter. The most obvious advantage of the Multiple Distribution algorithm, and the one that leads to the greatest expected performance improvement, is the fact that the algorithm generates fewer messages per job assignment than the Simple Distribution algorithm does. Even though the average message length may be longer, with a communications system like PVM, in which message set-up time tends to be the dominant cost in relatively short messages, decreasing the number of messages at the cost of increasing message length can improve overall performance. For example, it takes PVM a constant 3.9 milliseconds to set up a message, but only 0.5 microseconds per byte to pack and send it on an IBM RS/6000 model 560. Therefore, it will take the package 3.95 milliseconds to send a 100 byte message, or 4.4 milliseconds to send a 1000 byte message: the setup cost dominates message transmission time for short messages.

The Multiple Distribution algorithm has some other advantages as well. First, since Distributed NEPP's job compression system generates shorter messages for jobs with similar parameters, and since jobs with similar parameters tend to occur one after another in the input file, the Multiple Distribution algorithm's job assignment message tends to use fewer bytes per job than the Simple Distribution algorithm's does. Also, since the nodes tend to get groups of jobs that are similar in nature, they will tend to converge towards a result more quickly for each job, potentially improving Distributed NEPP's overall performance. In addition, the Multiple Distribution algorithm tends to maintain a job order for each node closer to the one in the input file, so the nodes will be more likely to converge to a result when they analyze an especially sensitive engine. Finally, the host does not have to wait until the node's job queue is empty before sending it a new set of jobs. The "minimum threshold size" parameter of the algorithm on page 10 allows the host to adjust how much work the node has left before the host sends it more. In Distributed NEPP, the host sends a block of jobs to a node whenever the node is working on the last job in its queue, avoiding the period in the Simple Distribution algorithm when the node is idle and waiting for a new job from the host.

Unfortunately, the Multiple Distribution algorithm is still not fault-tolerant. Just as with the Simple Distribution algorithm, if a node crashes, the host will distribute the remaining jobs and then hang forever waiting for the crashed node's results. Also, this algorithm can perform worse than the Simple Distribution algorithm in a heterogeneous environment because it increases the grain size from a single job to a group of jobs: with the Simple Distribution algorithm, the minimum time to finish the application is the time it takes the slowest node to execute 1 job. The minimum time for the Multiple Distribution algorithm is the time it takes the slowest node to finish a queue-full of jobs, so if one node runs very slowly, the whole application will have to wait for that node to finish.

3.1.4 Fault-Tolerant Distribution Algorithm

The *Fault-Tolerant Distribution* algorithm adds fault-tolerant aspects to the Multiple Distribution algorithm and solves the increased granularity problem. The basic idea of this algorithm is that it acts like the Multiple Distribution algorithm until it runs out of jobs in the input file. Then, the host begins sending replicas of unfinished jobs to the nodes. The nodes process the copied jobs just as they would any other job and return the results. The host, in turn, takes whichever of the replicated results arrives first and discards all other copies of those results. So it can keep track of which results it has, which it needs, and which it has replicated, the host mirrors each node's job queue internally. Whenever it gets the results of a particular job from a node, the host searches its internal queue structure for a matching job record. If the new results are the first set of results for that particular job, the host accepts them and marks all other copies of that job in its internal queues as REDUNDANT. Then, when the host receives the other, redundant copies of the job results, by consulting its internal queue structure it will know that those results are REDUNDANT and can discard them. Note that the node's portion of this algorithm is identical to the node's part of the Multiple Distribution algorithm. Here is the pseudocode for the host's part:

```
Algorithm: Fault-Tolerant Distribution, Host Portion
 begin
     broadcast design point and associated information
     calculate design point
     while (there are jobs left in the input file)
     begin
         (blocking) receive the next READY or DONE message from a node
         dequeue the corresponding entry from the host's queues
         if the host's queues show the results as REDUNDANT
```

```
            begin
                discard the results
            end
            else
            begin
                record the results
                search the host's queues, mark every other copy of those results as
                  REDUNDANT
            end
            if node's queue size is less than minimum threshold size
            begin
                pack an assignment message with enough jobs to fill the node's queue to
                  maximum threshold size
                send the assignment message to the node
                enter the jobs into the host's queues with type NORMAL
            end
        end
        while (the host has not received all results)
        begin
            (blocking) receive the next READY or DONE message from a node
            dequeue the corresponding entry from the host's queues
            if the host's queues show the results as REDUNDANT
            begin
                discard the results
            end
            else
            begin
                record the results
                search the host's queues, mark every other copy of those results as
                  REDUNDANT
            end
            if the node's queue size is less than the minimum threshold size
            begin
                select a set of jobs to duplicate
                pack the duplicate jobs into an assignment message
                send the assignment message to the node
                enter the jobs into the host's queues with type NORMAL
            end
        end
        broadcast a DIE message
    end
```

One or more node failures will not stop the Fault-Tolerant Distribution algorithm from finishing its computations: if a node fails before it has a chance to request a job, the host will send the work that node might have gotten to a different node instead. Furthermore, when the host reaches the end of the input file and begins replicating jobs, it will naturally replicate the work it sent to the failed node since that node will not have been able to respond with its results; the host will then receive the replica results from a healthy node and finish its computation. In fact, as long as the host and one node remain alive and have working communications links between them, this algorithm will be able to finish computation. Also, if a failed node or link comes back on line, as long as the recovered node signals its return with a READY message or the DONE message from its last assignment, the host could reintegrate it into the application. (Note, however, that Distributed NEPP does not yet include the capability to recover and reinitialize a crashed node as sections 3.3 and 6.1 explain. This limitation results more from lack of development time than from any limit of the Fault-Tolerant Distribution algorithm.)

The host, of course, is the system's Achilles' heel: if the host crashes the whole application will die. However, since Distributed NEPP is a single-user program, and since the engineer who uses it will probably run the host program from the workstation on her desk, a host crash in practical terms most likely means that a more serious problem occurred such as the workstation failing, in which case restarting Distributed NEPP is likely to be the least of the engineer's concerns. In addition, section 6.1 mentions some possible solutions for the host failure problem.

Another benefit of the Fault-Tolerant Distribution algorithm is that it decreases the minimum time the application takes to finish its work. Recall that for the Simple Distribution algorithm, the minimum time for the application to finish is the time it takes for the slowest node to complete one job, neglecting initialization and communication overheads. For the Multiple Distribution algorithm, the minimum time increases to the time it takes for the slowest node to finish all the jobs the host sent it in a single assignment message. Neither of these minima holds for the Fault-Tolerant Distribution algorithm: if one node is running so slowly that it holds up the whole application, the host will simply replicate any jobs remaining on the slow node, send them elsewhere for processing, and take whichever results come back first. In this case, the minimum time to complete the application is not entirely clear, but it does appear to be less than that of the Simple Distribution algorithm.

The main drawback to the Fault-Tolerant Distribution algorithm is that it increases the overall system load. Section 4.1 models the algorithm in more detail, but it is clear intuitively that as the number of job copies increases, nodes will tend to spend more of their time processing jobs that will eventually become redundant. In effect, the algorithm trades efficiency for speed and fault tolerance.

One issue the algorithm on page 11 glosses over is the process by which the host chooses a job to replicate. In general, the host uses some heuristics:

1. No node should ever have two replicas of the same job.
2. Replicate a job that is currently at the tail of a node's queue in preference to one at the head.
3. Replicate a job that currently has fewer replicas in the nodes' queues in preference to one that has more replicas.
4. When building a single assignment message that contains replica jobs, jobs that occurred next to each other in the input file should occur next to each other in the assignment message.

The NextJobToReplicate algorithm, below, will search the host's internal representation of the nodes' queues to find a candidate job to replicate using the above heuristics:

```
Procedure NextJobToRepicate (node, job, exclude, threshold)
Pass by reference: node, job, threshold
Pass by value: exclude
local variables: MinReplicas, AllRedundant, done, ReturnValue
 begin
     done = FALSE
     ReturnValue = SearchForCandidate (node, job, threshold, exclude, MinReplicas,
       AllRedundant)
     done = (ReturnValue is not NULL) or (AllRedundant is TRUE)
     if (not done)
      begin
           threshold = MinReplicas
           ReturnValue = SearchForCandidate (node, job, threshold, exclude, MinReplicas,
             AllRedundant)
     end
     return (ReturnValue)
end

Procedure SearchForCandidate (node, job, threshold, exclude, MinReplicas, AllRedundant)
pass by reference: node, job, MinReplicas, AllRedundant
pass by value: threshold, exclude
begin
    MinReplicas = infinity
    AllRedundant = TRUE
    done = FALSE
    StartNode = node
    StartJob = job
    while (done ≠ TRUE)
    begin
        if (job is not redundant)
          and (node ≠ exclude)
          and (exclude does not already have a copy of job)
        begin
            AllRedundant = FALSE
            MinReplicas = min (MinReplicas, this job's replicas)
            if (this job's replicas ≤ threshold)
            begin
                done = TRUE
                ReturnValue = job
            end
        end
        if (job ≠ tail of queue[node])
        begin
            job = NextJobCloserToTail(job)
        end
        else
        begin
            do
            begin
                node = (node + 1) mod NumNodes
            end
            while (node ≠ StartNode)
              and ((queues[node] is empty) or (node = exclude))
            job = head of queues[node]
        end
        if (node = StartNode) or (job = StartJob)
        begin
            ReturnValue = NULL
            done = TRUE
        end
    end
    return (ReturnValue)
end
```

Other procedures call NextJobToReplicate to decide which, if any, job they could replicate. The "node" and "job" arguments act as pointers to the next node and job NextJobToReplicate will examine to determine whether or not it can safely replicate that job according to the heuristics. Since NextJobToReplicate starts "job" at the tail of a given queue and increments it towards the queue's head, the routine implicitly satisfies heuristic 2. (Note the assumption that the procedure that first initializes "node" and "job" should initialize them to the tail of a queue.) NextJobToReplicate uses "threshold" to satisfy heuristic 3: the procedure first tries to find a candidate to replicate using the current threshold value. Only if it fails in its search does it increase the threshold value to the minimum threshold necessary to find a job to replicate. Other procedures use the "exclude" argument to make sure that NextJobToReplicate does not choose a candidate job from the queue of the node that is currently looking for replicas.

NextJobToReplicate calls SearchForCandidate to search the queue structure for a node to replicate. SearchForCandidate loops through the host's internal copy of the nodes' queues looking for a non-redundant job that satisfies the four heuristics. It has to handle two unusual cases: there may be no non-redundant jobs available, or it may be that every possible candidate has more replicas already in existence than the "theshhold" value. If there are no non-redundant jobs available, SearchForCandidate sets the "AllRedundant" flag. Also, it sets "MinReplicas" to the minimum number of replicas of any candidate job it encounters during its search so that NextJobToReplicate can raise the threshold if necessary. If, the first time NextJobToReplicate calls it, SearchForCandidate returns a value in "MinReplicas" greater than NextJobToReplicate's search threshold, NextJobToReplicate sets the current threshold equal to "MinReplicas" and calls SearchForCandidate again. In this way, NextJobToReplicate tends to return a job with the fewest possible copies for its caller to replicate.

3.2 Job Assignment Message Compression

When writing a parallel program, relatively small changes in the code's efficiency can translate to large changes in the efficiency of the overall application. According to Amdahl's Law [14], for p processors

$$S \leq \frac{1}{f_s + f_p/p} \tag{3.1}$$

where S is the speedup factor, f_s is the fraction of the parallel program's code that must run sequentially, f_p is the fraction that can run in parallel, and p is the number of processors available to the program. f_s represents a bottleneck in the parallel program: by decreasing it even slightly, the whole application benefits.

Distributed NEPP's job assignment compression scheme is designed to reduce the communications overhead of the host, the application's bottleneck. Since PVM has a high communications set up cost but relatively low cost per byte, reducing message size will not have as much of an effect on application run time and asymptotic speedup as reducing the number of messages, but it improves performance, nonetheless.

Job assignments consist of 7,803 separate variables: 4,800 single precision floating point numbers, 3,001 double precision floating point numbers, and 2 integers. However, typically very few of these elements typically change from one job to the next in the input file. Therefore, instead of sending all 7,803 elements, which would result in a message length of 43,216 bytes on the IBM RS/6000's that acted as Distributed NEPP's testbed, the host program sends only the difference between the node's previous job assignment and the current one. The message then consists of a sequence of "integer-new value" pairs, where the integer indicates which assignment element to change, and the "new value" specifies the new value of that element. This compression scheme reduces the size of a typical assignment to 3 or 4 double precision floating point numbers and an equal number of integers, for a message size of about 48 bytes per assignment.

To save memory, the host does not maintain an image of the most recent job assignment for each node. Instead, it keeps a single master job assignment image, and, for each node, a set of difference flags. When the host builds a new assignment message to send to a node, it sets a difference flag in every node's flag set for elements that differ between the new and previous assignments. Once it actually sends the assignment message to a node, the host clears that node's difference flags. Here is the pseudocode for the algorithm:

```
When creating an assignment:
begin
    for i = 0 to NumAssignmentElements - 1
    begin
        if MasterAssignmentImage[i] $\neq$ NewAssignmentImage[i]
        begin
            for j = 0 to NumberOfProcessors - 1
            begin
                DifferenceFlag[i,j] = TRUE
            end
            MasterAssignmentImage[i] = NewAssignmentImage[i]
            end
        end
        ElementCount = 0
        for i = 0 to NumAssignmentElements - 1
        begin
            if DifferenceFlag[i, DestinationNodeNumber] = TRUE
            begin
                ElementCount = ElementCount + 1
            end
        end
        pack ElementCount into assignment
        for i = 0 to NumAssignmentElements - 1
        begin
            if DifferenceFlag[i, DestinationNodeNumber] = TRUE
            begin
                pack i into assignment message
                pack MasterAssignmentImage[i] into assignment message
            end
        end
    end
    When sending one or more assignments to a node:
    begin
        for i = 0 to NumAssignmentElements - 1
        begin
            DifferenceFlag[i, DestinationNodeNumber] = FALSE
        end
    end
```

Each difference flag occupies only 1 byte, so the difference flag system requires only 18% as much memory per node as storing a whole image per node would need, at the expense of possibly sending an extra message element if the element changes to a different value and then changes back again between assignments to a particular node.

This message compression scheme provides even greater benefits under the Multiple Distribution and Fault-Tolerant Distribution algorithms than it does with the Simple Distribution algorithm. Since successive jobs in the input file tend to have fewer differences between their parameters, when the host packs several successive jobs into a single assignment message, the first assignment will typically be quite large, but the rest will tend to be smaller because of the similarities between successive jobs, resulting in an overall savings in message length.

It is worth noting that the algorithm on page 15 has to pack, along with each element it sends, the index of the differing element. As the number of differences increases, it will eventually become less expensive to send the entire contents of "NewAssignmentImage" rather than sending each individual element along with an integer describing which element it is. On the RS/6000, for example, double precision floating point numbers occupy 8 bytes, and single precision floating points and integer values occupy 4 bytes. If a message contains 4,800 single precision floating point numbers and 402 double precision floating point numbers, its total length will be 43,224 bytes, including the integer the algorithm packs along with each difference value. In this case, it would be less expensive for the algorithm to simply pack the whole image, suggesting that a simple optimization for the job compression algorithm would be for it to calculate, first, whether it would be less expensive to send the whole image instead, and, if sending the whole image would be cheaper, to pass "NumAssignmentElements" instead of "ElementCount" in the message to indicate that it contains a whole image instead of a set of differences and their indices.

The most important aspect of the job compression issue, however, is not the algorithm itself but the fact that by constructing messages intelligently, Distributed NEPP and other parallel programs can improve their efficiencies and cut communications overhead. In this case, as this section mentioned earlier, using this approach decreased the length of most assignment messages from 43,216 bytes to around 48.

3.3 Other Issues

3.3.1 Surviving a PVM Process Crash

One of the limitations of PVM version 2.4 is that if any machine in the cluster crashes, the whole application will crash along with it. The PVM message passing and remote process control facilities that Distributed NEPP uses rely on having one PVM daemon process running on each machine in the cluster. Under PVM version 2.4, however, if any PVM daemon in the cluster crashes, the other daemons will observe that fact and exit, in this case aborting Distributed NEPP as a consequence. PVM version 3.0, as [6] explains, detects a crashed daemon and automatically deletes it from the cluster. Version 3.0 also allows new machines to join the cluster dynamically, though Distributed NEPP does not yet have the capability of using a new machine if one should become available.

It is important to note that, while Distributed NEPP cannot survive a PVM daemon crash under PVM 2.4, it can survive a crash of one of its processing node programs. Furthermore, once the program has been modified to run under PVM 3.0, the Fault-Tolerant Distribution algorithm will handle daemon crashes just as it handles node crashes, by rerouting the work to still-functional nodes.

3.3.2 Node Recovery

Currently, Distributed NEPP does not include any facilities for restarting node processes or reintegrating processes that an external agent has somehow started into the application. However, there is no fundamental reason why the application might not be able to reintegrate node processes. Please see section 6.1 for some possible approaches to recovering nodes.

3.3.3 Advantages and Drawbacks of Centralized Control

As earlier sections mentioned, the host process is the vulnerable point in the Distributed NEPP system. If the host fails, the entire application will fail along with it. Furthermore, the host is the system bottleneck: as the number of node processes increases, the host's workload will increase as well, and eventually the host will saturate. For a number of real-world, application-specific reasons, however, centralized control is a viable alternative in this case.

First of all, Distributed NEPP is a single user application. Unlike the distributed systems many of the references describe, each user will start her own copy of the host program and will have her own set of node processes. If the host program dies, it will be just as though a uniprocessor application had died. The nodes will detect the fact that there is no host process when they try to send it a message and get an error result from PVM. When they discover the host is dead, they will automatically exit. The user need merely restart the host, just as she would with a uniprocessor application.

The assumption that only one machine in the cluster has access to the input and output files also lends itself to centralized control. If the only machine that can access the data files becomes faulty, there is no way for the application to continue. By placing the host, the only non-fault-tolerant portion of the Distributed NEPP application, on the only machine that can access the input and output files, the Distributed NEPP design consolidates two potential faults into one: if the critical machine should fail, the fact that the host will fail with it is not important since there is no way for the application as a whole to continue without access to its files.

One advantage of centralized control is that it avoids the additional overhead many of the algorithms in Chapter 2 need to gather information for load balancing. Since the host always knows which nodes have which jobs, it has all the information it needs to balance the system load without having to pay the communications overhead to gather information from throughout the system.

Another advantage of centralized control is that it is simple: converting Sequential NEPP to Distributed NEPP required adding over 15,000 lines of C code to the program; using a distributed scheme might have made the problem too large to solve with the time and resources available, and the benefits may not have been worth the extra effort.

Still, it should be possible in future versions of the application to incorporate a distributed control scheme. Section 6.1 lists some possible approaches to distributing the host's tasks throughout the processing cluster.

Chapter 4

Analytical Model

4.1 Analytical Model of the Algorithms' Performance

Without an analytical model, it would be extremely difficult to analyze the performance of the Simple Distribution, Multiple Distribution, and Fault-Tolerant Distribution algorithms in a heterogeneous environment. Since there are several users on the testbed machines, and system loads can vary widely from one test run to the next, traditional measures such as speedup show a large random variation. By developing a model that will calculate expected performance in terms of system load, and correlating the results of this model with observed performance, we can analyze the performance figures more accurately and meaningfully. Table 4.1 summarizes the notation this analytical model will use.

Consider a system containing p node processors. Let c_i represent the relative speed of processor number i's hardware. Now, in a multi-user system, the node processes may not have access to the full speed of the hardware, depending on the system load, so let α_i represent the fraction of the power of processor i that the application can use; α_i ranges from $0 \ldots 1$. Note that, while c_i is constant and depends on the processor's hardware, α_i depends on the current system load and, therefore, can change over time. To keep the model simple, define α_i to be the fraction of processor i's power that the application uses during the entire time the application runs.

Now, we can combine α_i and c_i to create a measure of the relative throughput for processor i. Let s_i represent the throughput, or speed, of processor i. Then, s_i will be:

$$s_i = \alpha_i c_i \tag{4.1}$$

Furthermore, define the average throughput of the nodes, s_μ as the average value of s_i across all processing nodes:

$$s_\mu = \frac{1}{p}\sum_{i=0}^{p-1} s_i \tag{4.2}$$

And finally, let s_{min} be the throughput of the slowest processor in the system:

$$s_{min} = \min\{s_i\}, \quad i = 0 \ldots p-1 \tag{4.3}$$

Similarly, a node must perform a certain amount of work (i.e. complete a certain number of standard instructions) equal to f_j on each job j to finish it. Note that this model assumes that a job is an atomic unit of computation: only one node may work on a particular job at a time, though several nodes may each work independently on their own copy of a given job. Also, each algorithm will introduce a certain amount of communication and computational overhead per job. Represent the algorithm-introduced overhead with a_j. Note that with some algorithms, a_j may be constant for all jobs, but for generality's sake, this model allows the overhead to vary from job to job. Now, define the total work a node must perform on a job to finish it as the sum of the work the job requires and the amount of overhead the job distribution algorithm introduces:

$$w_j = f_j + a_j \tag{4.4}$$

Table 4.1: Notation for algorithm performance model.

p	number of node processors in the system
q	queue size of each node processor (assuming uniform queue sizes among the processors), measured in jobs
J	number of jobs the host must distribute
f_j	the amount of computation it will take to finish job j, measured in standard instructions per job
a_j	the communications and processing overhead the algorithm introduces for job j, measured in standard instructions per job
w_j	the total amount of work job j requires, the sum of f_j and a_j, in standard instructions per job
w_μ	the average value of $w_j, \forall j$, in standard instructions per job
w_h	the overhead the host introduces per job result it receives, in standard instructions per job
c_i	the relative CPU and hardware speed of processor i, hardware dependent, measured in standard instructions per unit time
α_i	fraction of the CPU time of processor i that the application receives during the complete run, unitless
s_i	the relative throughput (speed) of processor i; the product of c_i and α_i, in standard instructions per unit time
s_μ	average value of $s_i, \forall i$, in standard instructions per unit time
s_{min}	minimum of $s_i, \forall i$, in standard instructions per unit time
T_{simp}, T_{mult}, T_{fault}	wall clock times for the Simple Distribution, Multiple Distribution, and Fault-Tolerant Distribution algorithms, respectively
t_m, t_r, t_l	time for Multiple Distribution, Replicating, and Last Job phases, respectively, of the Fault-Tolerant Distribution algorithm
W_{simp}, W_{mult}, W_{fault}	total amount of computation (aggregate CPU time) the Simple Distribution, Multiple Distribution, and Fault-Tolerant Distribution algorithms perform, respectively, in standard instructions
x	represents an arbitrary number of jobs

Furthermore, let w_μ represent the amount of work a given job requires on average:

$$w_\mu = \frac{1}{J}\sum_{j=0}^{J-1} w_j \tag{4.5}$$

The total computational power in the nodes will be $\sum_{i=0}^{p-1} s_i$ which is equal to ps_μ. Furthermore, if, on average, the amount of work a node must perform to finish a job is w_μ, then the host can expect

$$\textit{responses per time} = \frac{ps_\mu}{w_\mu} \tag{4.6}$$

job results from the nodes per unit time, or, equivalently, the mean time between responses from the nodes will be

$$\textit{time per response} = \frac{w_\mu}{ps_\mu} \tag{4.7}$$

Consider the Multiple Distribution algorithm first. As Chapter 3 describes the algorithm, the programmer can configure both the number of jobs the host sends to a node with each assignment message and how small the number of jobs in a node's queue can get before the host regards that node as idle. For the sake of simplicity, however, this model assumes that the host regards a node as idle when the node empties its queue, at which point the host refills the node's queue. This assumption should not prove especially limiting since the model does not include specific terms for communications overhead, and it assumes that the host may refill queues asynchronously. Therefore, if, for example, the host actually sends five jobs per assignment message and regards the nodes as idle when their queues contain one job, the model will treat the system as queue size 6 with adjustments to some constants to reflect the difference in communications overhead.

Under the Multiple Distribution algorithm, the host sends a total of J jobs to the nodes and does not send replica jobs. Therefore, under normal circumstances, we would expect the application to finish in time

$$t_{mult-ave} = \frac{Jw_\mu}{ps_\mu} \tag{4.8}$$

However, there is also a lower bound on the time for this algorithm. At the beginning of the algorithm's run, the host sends enough jobs to every node to fill the nodes' queues (assuming $J \geq pq$.) Therefore, the minimum time for the algorithm's run is the time it takes the slowest node to finish all the jobs in its queue, or, on average,

$$t_{mult-min} = \frac{qw_\mu}{s_{min}} \tag{4.9}$$

The total time for the Multiple Distribution algorithm will be the larger of $t_{mult-ave}$ and $t_{mult-min}$; if the nodes are evenly balanced, they will all finish simultaneously, but if one node is significantly slower, the application will have to wait for it to finish with its assigned work:

$$T_{mult} = \max\left\{\frac{Jw_\mu}{ps_\mu}, \frac{qw_\mu}{s_{min}}\right\} \tag{4.10}$$

The total time for the Simple Distribution algorithm is a special case of that for the Multiple Distribution algorithm. The Simple Distribution algorithm acts like a Multiple Distribution algorithm with queue size 1 and higher communication overheads, yielding the equation

$$T_{simp} = \max\left\{\frac{Jw_\mu}{ps_\mu}, \frac{w_\mu}{s_{min}}\right\} \tag{4.11}$$

in which the value of w_μ will be somewhat larger than that in equation 4.10 since a_j, the amount of overhead the algorithm introduces to each job, will be larger.

The analysis for the Fault-Tolerant Distribution algorithm is more complex. Recall that the algorithm acts like the Multiple Distribution algorithm until it exhausts all the jobs in the input file. Then, as nodes become idle, the host begins sending replicas of jobs whose results it has not yet received to the idle nodes.

Furthermore, jobs can become redundant: once the host has received one copy of the results for a given job, all other copies of that job in the nodes' queues become redundant, and the host simply discards the results when the nodes send them. Since the Fault-Tolerant Distribution algorithm as presented in Chapter 3 does not include any provisions for dequeuing redundant jobs, as the run continues the nodes will waste increasing amounts of their time processing now-redundant information. (For practical reasons, the Distributed NEPP code does not dequeue redundant jobs. However, Chapter 6 includes a discussion of the merits of this refinement.)

Assume $J \geq pq$. The host tries to keep the nodes' queues full and sends out new jobs in preference to replicas. Therefore, when the host sends the last job from the input file it will have received the results of $J - pq$ jobs from the nodes, since the nodes' queues still hold a total of pq jobs for them to process. Furthermore, none of the $J-pq$ job results the host receives will be redundant because, up to that point, the host has not sent any replica jobs. So, for the first $J-pq$ jobs, the Fault-Tolerant Distribution algorithm acts like the Multiple Distribution algorithm. Call this time period the *Multiple Distribution* phase. Substituting into equation 4.8 yields the time the Fault-Tolerant Distribution algorithm spends in this phase:

$$t_m = \frac{(J - pq)w_\mu}{ps_\mu} \tag{4.12}$$

During the next phase of the algorithm, the *Replicating* phase, the host begins replicating jobs and sending the replicas to the nodes. Also during this phase, the host begins receiving and discarding redundant job results. Let x be the number of original, non-redundant job results that the host needs to finish the application. Since the host keeps the nodes' job queues full by sending them replica jobs, the fraction of non-redundant results still in the nodes' queues will be x/pq. The rest of the jobs in the queues are or will become redundant information. (Note that if there are several copies of a particular job in the nodes' queues, and the host has not yet received the results for that particular job, we have no way of knowing which of the copies is redundant and which is non-redundant; we can, however, state that one of the copies will be non-redundant and the rest will hold redundant information when the host receives them.) Here the model makes another simplifying assumption: that the redundant and non-redundant jobs have uniform distribution in the nodes' queues. In other words, the model assumes that if there are x/pq non-redundant jobs in the queues, then the next job result the host receives will be non-redundant with probability x/pq (i.e. $P_{nr}(x) = x/pq$). Furthermore, each non-redundant result the host receives decreases the number of results it needs by one. Finally, equation 4.6 gives the expected number of responses, both redundant and non-redundant, the nodes will send to the host per unit time. We can combine this information into an equation that says that the rate of change of the number of non-redundant jobs in the queues will equal the number of non-redundant jobs the nodes send to the host at each time instant, or:

$$\frac{dx}{dt} = -\left(\frac{x}{pq}\right)\left(\frac{ps_\mu}{w_\mu}\right) \tag{4.13}$$

Solving equation 4.13 gives

$$\ln x = -\left(\frac{s_\mu}{qw_\mu}\right)t + K \tag{4.14}$$

where K is the constant of integration. At time 0, all the jobs in the nodes' queues are non-redundant, so x should be pq, implying that $K = \ln pq$. Plugging in the value for K, and solving for t, yields the following equation:

$$t = \left(\frac{qw_\mu}{s_\mu}\right)(\ln pq - \ln x) \tag{4.15}$$

Equation 4.15 holds only until $x = q$. If there are pq jobs in the nodes' queues, x of which are non-redundant, then the number of copies of each non-redundant job will be pq/x. (Note that, for simplicity's sake, this term neglects that fact that some of the jobs in the queues will not be copies of jobs for which the host still needs results. In other words, in reality the actual number of copies of each of the x jobs the host still needs will be slightly less than pq/x.) Now, if $x < q$, the number of copies of each job the host needs will be $pq/x > p$, indicating that the host has put more than p copies of at least one of the jobs it needs into the nodes' queues. However, with the Fault-Tolerant Distribution algorithm, the host will never send two copies of the same job to the same node; if sending a node multiple copies of the same job is the host's only option, the host will leave the node's queue partially empty, instead. Therefore, once $x = q$, the host cannot send out any more copies of any of the jobs it needs, so the Replicating phase ends making the total time for the Fault-Tolerant Distribution algorithm's Replicating phase

$$t_r = \left(\frac{qw_\mu}{s_\mu}\right)(\ln pq - \ln q) = \left(\frac{qw_\mu}{s_\mu}\right)(\ln p) \tag{4.16}$$

During the final, *Last Job* phase of the algorithm, the host cannot replicate any jobs without violating the "one copy of a job to a node" rule, because there are so few original results it needs. At the beginning of this phase, the total number of jobs, redundant and non-redundant, in the nodes' queues will be pq. The host can expect ps_μ/w_μ job responses per unit time, from equation 4.6, so the total number of jobs in the queues at any given time during this phase will be:

$$jobs\ in\ queues = pq - \left(\frac{ps_\mu}{w_\mu}\right)t \tag{4.17}$$

leading to a new version of equation 4.13 which describes how the number of non-redundant jobs will change during this time period:

$$\frac{dx}{dt} = -\left(\frac{x}{pq - \frac{ps_\mu}{w_\mu}t}\right)\left(\frac{ps_\mu}{w_\mu}\right) \tag{4.18}$$

Solving equation 4.18 yields

$$x = K(qw_\mu - s_\mu t) \tag{4.19}$$

Where K is the constant of integration. Since the Last Job phase of the Fault-Tolerant Distribution algorithm begins when $x = q$ at $t = 0$, we can calculate $K = 1/w_\mu$. Plugging in the value for K and solving for t gives equation 4.20, below:

$$t = \frac{w_\mu}{s_\mu}(q - x) \tag{4.20}$$

Finally, the Last Job phase ends when there are no non-redundant jobs left in the queues, i.e. when $x = 0$. When $x = 0$, equation 4.20 provides an expression for the time the Fault-Tolerant Distribution algorithm takes to complete its Last Job phase:

$$t_l = \frac{qw_\mu}{s_\mu} \tag{4.21}$$

If we combine equations 4.12, 4.16, and 4.21, we have an expression for T_{fault}, the total time for the Fault-Tolerant Distribution algorithm to run:

$$T_{fault} = \frac{(J - pq)w_\mu}{s_\mu p} + \left(\frac{qw_\mu}{s_\mu}\right)(1 + \ln p) = \frac{Jw_\mu}{ps_\mu} + \left(\frac{qw_\mu}{s_\mu}\right)\ln p \tag{4.22}$$

Another statistic of interest is the total amount of computation all three algorithms must perform. Since the Simple Distribution and Multiple Distribution algorithms do not have the nodes perform any redundant work, calculating their total computation is straightforward. The total work the application requires is

$$\sum_{j=0}^{J-1}(w_j + w_h) = J(w_\mu + w_h) \tag{4.23}$$

where w_j includes both the work each job of the application requires and the overhead the algorithm introduces, and w_h is the amount of overhead the host introduces per job result it receives from the nodes. Since the Simple Distribution and Multiple Distribution algorithms do no redundant computation, their total computation, W_{simp} and W_{mult}, respectively, will be

$$W_{simp} = W_{mult} = J(w_\mu + w_h) \tag{4.24}$$

Where the values of w_μ and w_h will be different for each of the two algorithms due to different overhead.

The Fault-Tolerant Distribution algorithm will require more total computation because the nodes spend a portion of their time calculating redundant results. We can find the work the nodes perform by multiplying the work per job by the jobs per unit time by the total algorithm run time:

$$W_{fault-nodes} = (w_\mu)\left(\frac{ps_\mu}{w_\mu}\right)(T_{fault}) \tag{4.25}$$

Furthermore, the host will require some CPU time. The host's total contribution to the application's computational load will be the work it requires per job result multiplied by the number of job results it receives, or

$$W_{fault-host} = (w_h)\left(\frac{ps_\mu}{w_\mu}\right)(T_{fault}) \tag{4.26}$$

The total workload the application requires under the Fault-Tolerant Distribution algorithm will be

$$W_{fault} = W_{fault-nodes} + W_{fault-host} \tag{4.27}$$

which, after substituting and simplifying, yields the equation

$$W_{fault} = (w_\mu + w_h)[J + pq(\ln p)] \tag{4.28}$$

where $(w_\mu + w_h)pq\ln p$ is the additional amount of computational overhead the Fault-Tolerant Distribution algorithm introduces through having the nodes compute redundant information. Note that here, too, the values of w_μ and w_h will be different from those of the Simple Distribution or Multiple Distribution algorithms.

4.2 Corrected Efficiency

Measuring execution time, speedup, and efficiency in a heterogeneous, multi-user environment with dynamically changing loads presents some difficulties. Execution time and speedup are both important figures for the end user, so it makes sense to measure them in the environment where the end user will be working and to present an average figure over several trials so that the user knows what sort of performance to expect from the application. Efficiency, on the other hand, is a figure of more importance to the application's designer: it represents how well the designer has done at minimizing overhead while creating a parallel application. Therefore, it may be worth considering a measure of efficiency that is independent of the operating environment.

When engineers measure the efficiency of an electric motor, they calculate the amount of power at the output of the motor divided by the power they must supply at the input, and they make similar measurements of other mechanical systems. An intuitively appealing measure of a parallel program's efficiency, then, would be the amount of work it accomplishes divided by the amount of work the multiprocessor must supply to the application, or, more specifically, the number of standard CPU cycles of useful work the application finishes divided by the number of cycles it consumes.

Consider an application similar to Distributed NEPP. Suppose the application must perform J jobs, each requiring work f_j. Furthermore, suppose the parallel version of the application generates an overhead equal to $A(J)$. (This analysis uses the symbol "A" for the overhead function because it is similar in concept to the overhead term, "a_j", that section 4.1 introduced. Table 4.2 summarizes the additional notation this section uses.) The total amount of work the program must perform to finish all J jobs will be:

$$W_{req} = \sum_{j=0}^{J-1} f_j \tag{4.29}$$

And the work that the parallel program will perform will be:

$$W_{par} = A(J) + \sum_{j=0}^{J-1} f_j \tag{4.30}$$

Table 4.2: Additional notation section 4.2 introduces.

$A(J)$	additional overhead the parallel application introduces, in standard instructions
W_{req}	amount of work the application must do, exclusive of additional parallel overhead, in standard instructions
W_{par}	total number of standard instructions the parallel application executes, including overhead
T_{seq}	time for sequential version of application to finish execution
T_{par}	time for parallel version of application to finish execution
S	speedup: T_{seq}/T_{par}, unitless
E	normal measure of efficiency: S/p, unitless
E_c	corrected efficiency: W_{req}/W_{par}, unitless
α_s	time slice the sequential program gets on the test processor, unitless
c_s	relative speed of the sequential processor, in standard instructions per unit time

We can then define the *corrected efficiency*, the efficiency of the application that takes into account the relative speeds of the various CPUs in the multiprocessor and how much time the application gets on each one, as:

$$E_c = \frac{W_{req}}{W_{par}} = \frac{\sum_{j=0}^{J-1} f_j}{A(J) + \sum_{J=0}^{J-1} f_j} \tag{4.31}$$

In practical use, the engineer could measure W_{req} and W_{par} in terms of CPU seconds that the sequential and parallel versions of the program required, for the given number of nodes, and corrected for the relative speeds of the sequential and parallel CPUs.

Now, for comparison, consider the more standard definition of efficiency. For the sake of generality, this analysis will allow the application to run in a heterogeneous, multi-user environment. In particular, let c_i be the relative speed of each of the i processors the application uses, and let α_i be the fraction of time the application has access to the CPU on processor i, where we assume that, since there are multiple users, the various users are sharing the different processors to one degree or another. We also define α_s and c_s to be the time fraction and relative speed, respectively, of the processor on which the sequential version of the program runs. The sequential program should finish running in time:

$$T_{seq} = \frac{\sum_{j=0}^{J-1} f_j}{\alpha_s c_s} \tag{4.32}$$

where we assume that any overhead the sequential program introduces is incorporated into f_j, as most measurements of speedup and efficiency do.

The parallel version of the program will finish in an amount of time equal to total work plus overhead divided by total processing power available, where the overhead might vary depending on the arrangement of jobs on the different processors:

$$T_{par} = \frac{A(J) + \sum_{j=0}^{J-1} f_j}{\sum_{i=0}^{p-1} \alpha_i c_i} \tag{4.33}$$

Speedup is equal to T_{seq}/T_{par}, and the usual definition of efficiency is speedup divided by the number of processors:

$$E = \frac{S}{p} = \frac{\frac{\sum_{j=0}^{J-1} f_j}{\alpha_s c_s}}{p\frac{A(J)+\sum_{j=0}^{J-1} f_j}{\sum_{i=0}^{p-1} \alpha_i c_i}} \tag{4.34}$$

or, rewriting for clarity,

$$E = \left(\frac{\sum_{j=0}^{J-1} f_j}{\alpha_s c_s}\right)\left(\frac{\frac{1}{p}\sum_{i=0}^{p-1} \alpha_i c_i}{A(J)+\sum_{j=0}^{J-1} f_j}\right) \tag{4.35}$$

Now, consider the special case of a homogeneous, single user multiprocessor. If the machine has only one user, or, more accurately, only one application running on any given node at a time, then that parallel program will not have to share the processor's nodes, so $\alpha_i = 1$ for all i. Furthermore, since the machine is homogeneous, all processors will have the same speed, so $c_i = 1$ for all i. Finally, suppose the user follows the usual practice of testing the sequential version of the code on one processor of the parallel machine to get an accurate time for speedup calculations. Then, $\alpha_s = c_s = 1$ as well, and equation 4.35 simplifies to:

$$E = \frac{\sum_{j=0}^{J-1} f_j}{A(J)+\sum_{j=0}^{J-1} f_j} \tag{4.36}$$

which is exactly the same as equation 4.31. So, in the limiting case of a homogeneous, single-user multiprocessor, the corrected efficiency and the conventional definition of efficiency are identical. One concern at this point might be the fact that conventional efficiency on a single-user multiprocessor includes the time during the application run when the processors are idle, such as during the sequential portion of the parallel code, and no explicit term for this idle time appears in equation 4.36. However, no other application can use the processors during their idle periods, since only 1 application can use the node at a time in a single-user machine, so we can incorporate the idle times into $A(J)$.

The two measures of efficiency are not the same in a more dynamic environment, however. Suppose a programmer is testing a new application on a homogeneous, multi-, as opposed to single-, user machine. She measures the sequential program on one node and gets a time. Then, when she runs the parallel version of the code, another programmer tests a different application at the same time, so both programmers get 1/2 the cycles of each processing unit. The corrected efficiency, E_c, will not change: each parallel program does the same amount of work, regardless of how long it takes. (And, since all processors run at the same speed, the programmer could calculate corrected efficiency using the number of CPU seconds the sequential and parallel versions of her program consumed.) However, the normal measure of efficiency will change dramatically. Since the programmer measured the sequential version of her code on a single processor and got the whole processor to herself, $\alpha_s = 1$. However, she is sharing the parallel processors, so $\alpha_i = 0.5$ for all i. Finally, since this hypothetical machine is a homogeneous multiprocessor, $c_i = 1$, and $c_s = 1$ because she tested the sequential program on one of the processing nodes. The normal method of measuring efficiency, dividing the speedup by the number of processors, will yield, by substitution into equation 4.35:

$$E = \left(\frac{\sum_J f_j}{1}\right)\left(\frac{0.5}{A(J)+\sum_J f_j}\right) \tag{4.37}$$

or 1/2 of what the programmer would have measured had she had sole use of the machine during testing. Furthermore, her efficiency measurements will vary with whatever the current load happens to be on the machine at the time, making it extremely difficult to tell how much overhead the parallel version of the code has introduced.

The normal measure of efficiency also varies in a heterogeneous multiprocessor. Suppose the engineer above was testing her code on a machine with mixed processors, so that half the processors had relative speed 1.0 ($c_i = 1.0$ for $i = 0 \ldots p/2-1$), and the others ran half as fast ($c_i = 0.5$ for $i = p/2 \ldots p-1$.) Again, the corrected efficiency will not change. However, the normal measure of efficiency will yield

$$E = \left(\frac{\sum_J f_j}{c_s}\right)\left[\frac{\sum_{i=0}^{p/2-1} 1.0 + \sum_{i=p/2}^{p-1} 0.5}{p\,(A(J)+\sum_J f_j)}\right] \tag{4.38}$$

assuming the user gets the whole machine in both cases ($\alpha_i = \alpha_s = 1.0$). If we further assume that our hypothetical engineer tests the sequential code on one of the faster processors ($c_s = 1.0$), then equation 4.38 simplifies to

$$E = \frac{3}{4}\left(\frac{\sum_J f_j}{A(J) + \sum_J f_j}\right) \tag{4.39}$$

or 3/4 what she would have measured on a homogeneous machine. Had she measured the sequential code on one of the slow processors, she would have found an overall efficiency of 150% that of the homogeneous case.

In both these cases, measuring the efficiency by dividing speedup by the number of processors results in a figure that includes the current system load and is sensitive to variations in hardware. While this measure of efficiency would be useful for someone who wants to know how much speedup to expect by adding new processors under average load conditions, it is less useful for determining $A(J)$, the amount of overhead a parallel program requires, in a homogeneous, multiuser environment. Therefore, later chapters present results both in terms of the usual measure of efficiency, speedup divided by number of processors, and in terms of corrected efficiency.

It is worth noting that, according to the model in section 4.1, Distributed NEPP's corrected efficiency should be the work the sequential program requires divided by W_{fault}. Dividing equation 4.29 by equation 4.28 yields

$$E_c = \frac{\sum_J f_j}{(w_\mu + w_h)[J + pq \ln p]} \tag{4.40}$$

which we can rewrite as

$$E_c = \frac{\sum_J f_j}{\sum_J f_j + \sum_J a_j + Jw_h + \frac{pq\ln p}{J}\left[Jw_h + \sum_J (a_j + f_j)\right]} \tag{4.41}$$

to see that, for Distributed NEPP using the Fault-Tolerant Distribution algorithm, the overhead term from equation 4.31 is equal to

$$A(J) = \sum_J a_j + Jw_h + \frac{pq\ln p}{J}\left[Jw_h + \sum_J (a_j + f_j)\right] \tag{4.42}$$

Furthermore, the Distributed NEPP program itself automatically collects and returns data on CPU usage from the host and all nodes, to provide real-world numbers to compare against the model.

4.3 Examination of the Analytical Model's Implications

The analytical model that section 4.1 develops implies a number of interesting things about the Simple Distribution, Multiple Distribution, and Fault-Tolerant Distribution algorithms. Perhaps the most interesting is that a heterogeneous environment can favor higher-overhead algorithms, such as the Fault-Tolerant Distribution or Simple Distribution algorithms, over those like the Multiple Distribution algorithm which have lower overhead requirements but a potentially higher minimum execution time.

Consider a cluster that contains 19 processors that run at relative speed 1.0 and 1 processor that runs much more slowly, at relative speed 0.2. Such a situation is actually fairly common: in one of the clusters that provides a testbed for Distributed NEPP, the relative speeds of the node processors range from 1.0 to 0.36 before including the effects of having to share the processors among multiple users. Furthermore, let the average computation per job, w_μ, be 1.0 for all three algorithms. Assume the Multiple Distribution and Fault-Tolerant Distribution algorithms use a queue size of 6, and that there are 100 jobs in the input file. Figure 4.1 shows the execution time the model predicts for each possible number of nodes, assuming that the slow processor is always one of the nodes. Note that the time for the Multiple Distribution algorithm, the dashed line, reaches a minimum at 4 processors. At this point, the fast processors are waiting for the slow one to finish the jobs the host has assigned it. The dotted line shows the performance of the Simple Distribution algorithm: since the minimum time for the Simple Distribution algorithm is so much lower,

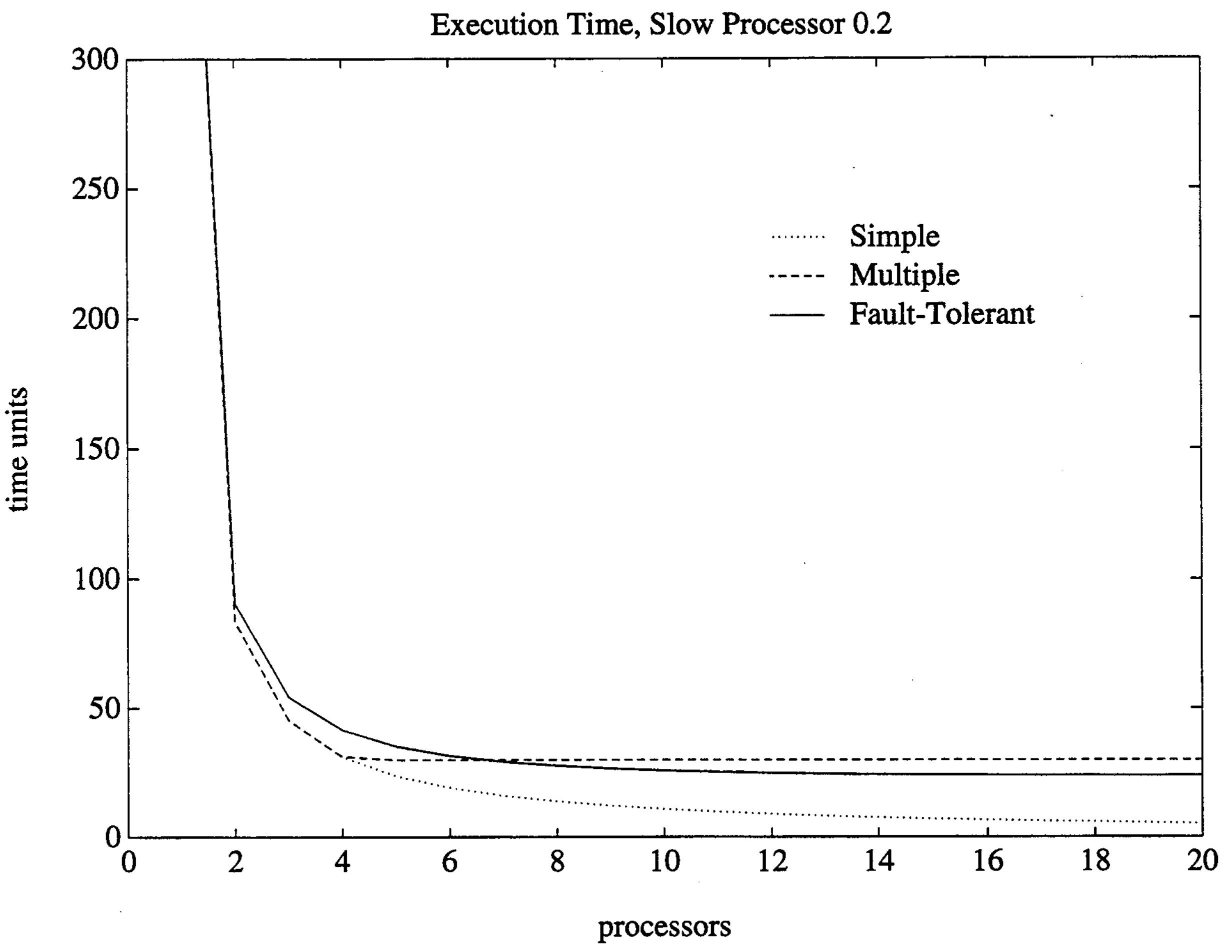

Figure 4.1: Predicted execution time, one slow processor at relative speed 0.2.

its performance continues to improve as the number of processors increases. Finally, the Fault-Tolerant Distribution algorithm, with the solid line, tracks the Simple Distribution algorithm, but its extra overhead keeps its performance lagging behind that of its simpler counterpart.

Figure 4.2 shows a somewhat more extreme case. Here, the speed of the slowest node has dropped to 0.03; all other factors remain the same. The Multiple Distribution algorithm, with it's 6-element queue, reaches its minimum execution time almost immediately. Even the Simple Distribution algorithm hits a plateau fairly quickly, so that, beyond 6 processors, only the less efficient Fault-Tolerant Distribution algorithm continues to show improving execution times.

It is important to note that these execution time plots can be somewhat deceptive in that they assume the same overhead per job for all three algorithms. As section 4.1 explains, these overheads should vary. In particular, the Simple Distribution algorithm should suffer from a higher overhead per job because of its greater communication requirements: it sends only one job with each assignment message rather than more efficiently packing several jobs into an assignment, as the Multiple Distribution and Fault-Tolerant Distribution algorithms do.

Speedup and efficiency follow the same trends as the execution times. Figure 4.3 shows the speedup values that correspond to the times in figure 4.2, measured against the same application running sequentially on a single, fast node. Here, again, the dotted line shows the performance of the Simple Distribution algorithm, the dashed line shows the Multiple Distribution one, and the solid line shows the Fault-Tolerant Distribution algorithm. Both the Simple Distribution and Multiple Distribution algorithms quickly reach peak speedup

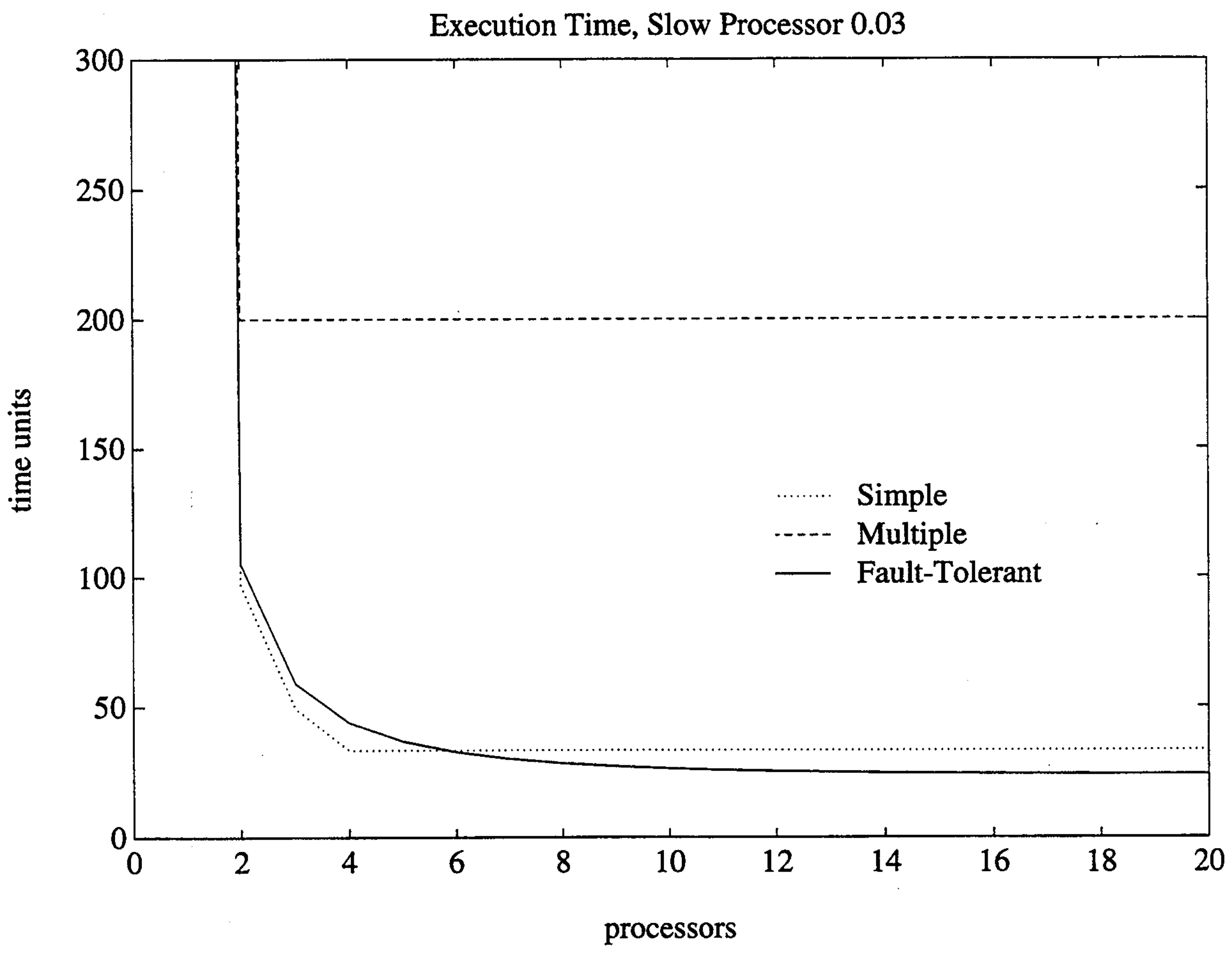

Figure 4.2: Predicted execution time, one slow processor at relative speed 0.03.

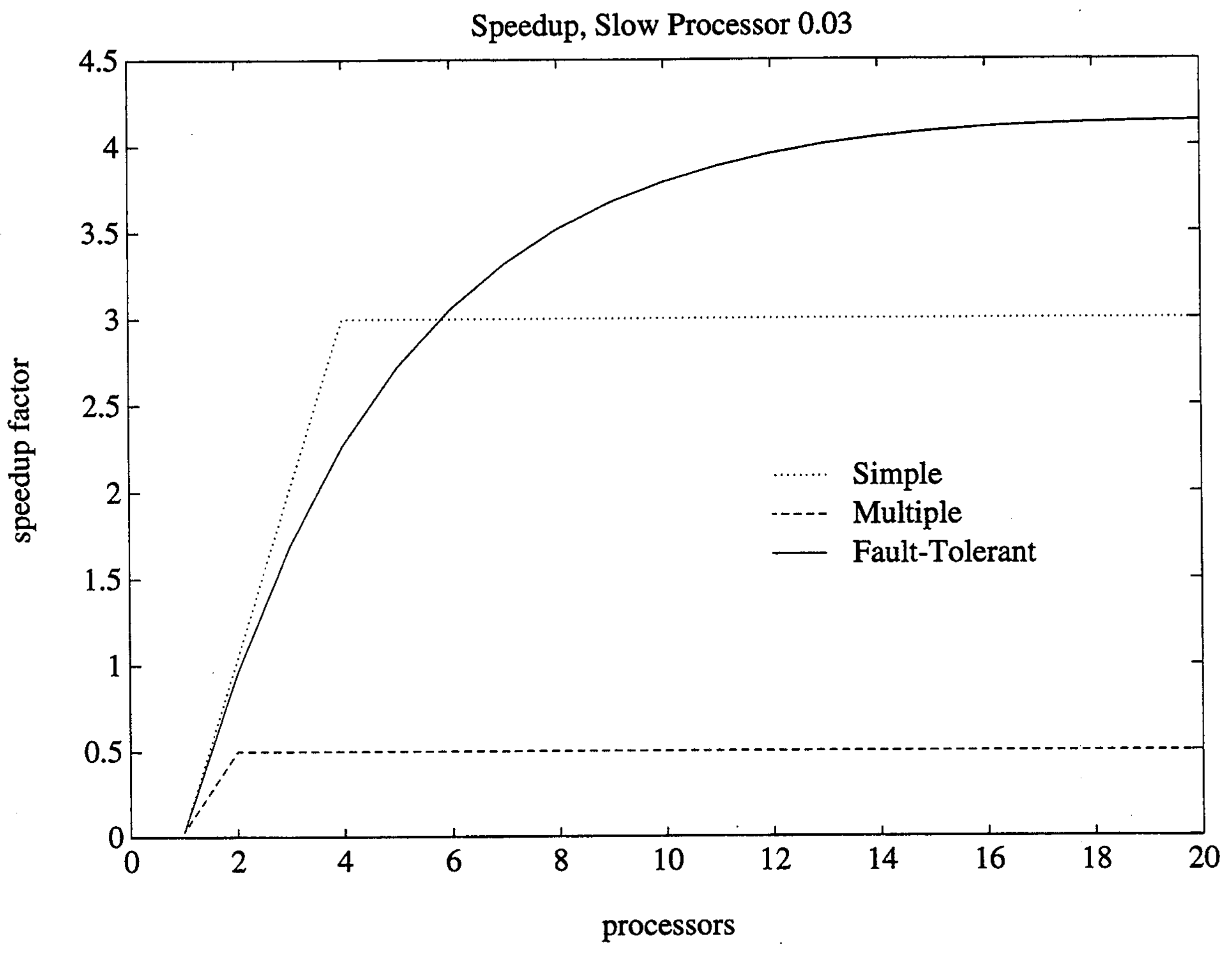

Figure 4.3: Predicted speedup, one slow processor running at relative speed 0.03.

and then level off as the slow node dominates computation time. The Fault-Tolerant Distribution algorithm, because it is less efficient and requires the nodes to compute redundant information, does not increase in speed with the number of processors as quickly as the Simple Distribution algorithm does, but it achieves a higher ultimate speedup factor because it does not face the same minimum execution time barrier that the other two do. The odd value for speedup with one processor is due to the fact that, for simplicity, this example assumes the slow node is always one of the processors in the group; with only one processor, the slow node will run the whole application, resulting in excessive execution times. Figures 4.1 and 4.2 do not show the high execution time for the 1 node case because they have been scaled to show details at two or more nodes.

Figure 4.4 shows the efficiency curves corresponding to figures 4.2 and 4.3, where efficiency equals speedup divided by the number of processors. The plateau in the execution times of the Simple Distribution and Multiple Distribution algorithms comes across very clearly in their efficiency curves, so that, when there is one very slow node and the rest of the nodes are operating at full speed, the Fault-Tolerant Distribution algorithm is ultimately more efficient than the other two. Again, the extremely low efficiency value in the 1 processor case is due to this example's assumption that if the application uses only one processor, it will be the slow one.

Finally, figure 4.5 shows the corrected efficiency curves for this same case. Since these curves are independent of execution time, measuring, instead, the impact each algorithm has on the system as a whole, they do not show the same glitch at 1 processor that figures 4.3 and 4.4 do. The Simple Distribution and Multiple

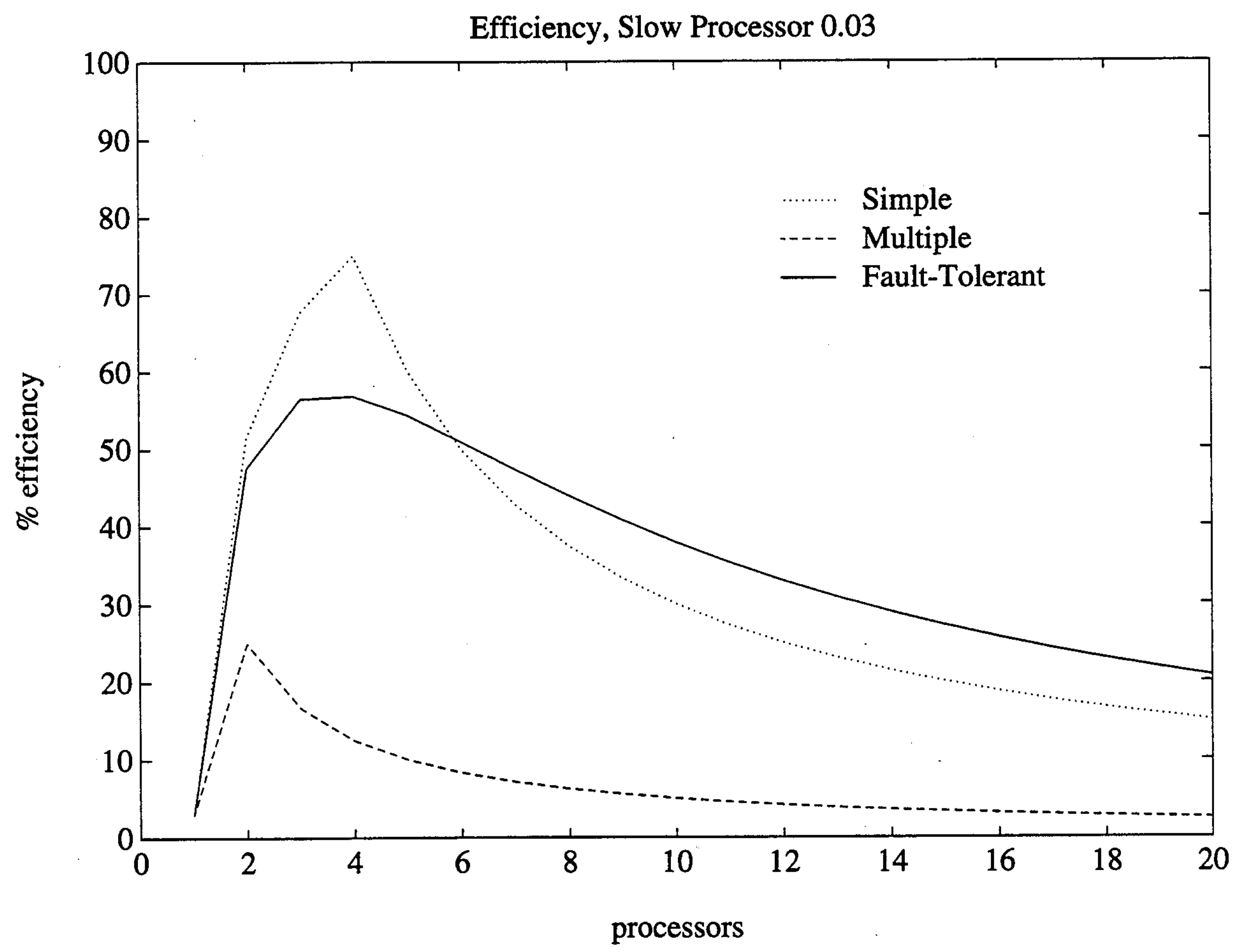

Figure 4.4: Predicted efficiency, one slow node running at speed 0.03.

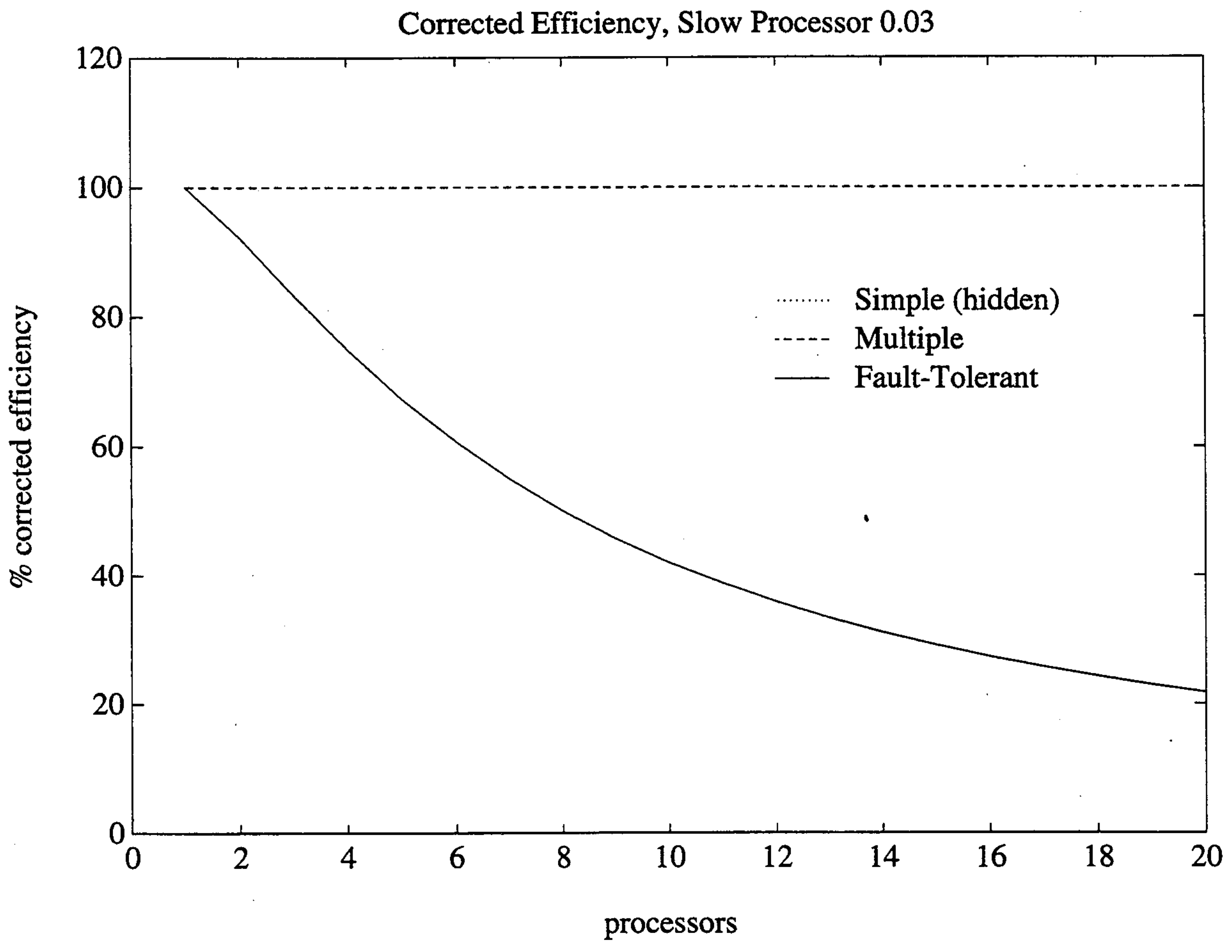

Figure 4.5: Predicted corrected efficiency, one slow node running at speed 0.03.

Distribution corrected efficiency curves, with the dashed line, will not change since, regardless of the number of processing nodes, both algorithms will do the same amount of work. Please note that the dotted line of the Simple Distribution algorithm is hidden behind the dashed Multiple Distribution algorithm's line in this graph.

As the number of processors increases, the Fault-Tolerant Distribution algorithm does more redundant work, so its corrected efficiency curve drops: with increasing processors, it consumes more CPU cycles on the processing nodes per unit work it accomplishes. This is the behavior equation 4.40 predicts. One final caveat about this figure is that, for clarity, the curves assume that the host introduces no overhead per job (i.e. $w_h = 0$); including an overhead factor for the host would change the values in the plots, but their overall shapes would remain the same.

Chapter 5

Results and Analysis

5.1 PVM 2.4 Times

Timing runs of a simple program using PVM version 2.4 to communicate between two IBM RS/6000 Model 550 computers indicate that PVM has a high set-up cost and low cost per byte, just as earlier chapters mentioned. For two processes on the same machine, the average message set-up time was 3.2 milliseconds. The set-up time rose to 3.9 milliseconds for two processes located on different workstations connected by an Ethernet network. The cost per byte was 0.22 microseconds to send a message to a process on the same machine, or 0.50 microseconds per byte to send to a process on a different machine. Therefore, on the RS/6000 model 550, it should take $3.2 \times 10^{-3} + b(2.2 \times 10^{-7})$ seconds to send a message of b bytes to another process on the same machine, or $3.9 \times 10^{-3} + b(5.0 \times 10^{-7})$ seconds to send the message to a process on a different machine. Note that the analytical model in Chapter 4 incorporates these times into the constants a_j, w_h, α_i, and α_h. a_j includes the number of standard instructions the node uses to communicate with the host (or the ones it spends idling while waiting to acquire a communications channel, if no other process can access the CPU during that time) along with all other overhead per job the algorithm introduces, and w_h functions similarly for the host. If another process can use the CPU during most of the message set-up time, then the set-up time instead affects α_i and α_h, the fraction of the application's execution time the host and node spend running on the CPU.

5.2 Distributed NEPP's Performance

The first test of Distributed NEPP determined its relative performance under the Simple Distribution, Multiple Distribution, and Fault-Tolerant Distribution algorithms. For this test, Distributed NEPP ran on a cluster of 32 IBM RS/6000 model 560 computers connected by an Ethernet network. Several users shared the cluster, so system loads tended to vary significantly over time; accordingly, these results show the mean values over multiple timing runs (10 runs for the Fault-Tolerant Distribution algorithm, 5 each for Multiple Distribution and Simple Distribution.) These trials used a fairly complex input file: the sequential version of NEPP required approximately 2171 seconds (36 minutes) wall clock time to process this particular file on an unloaded processor.

Figure 5.1 shows the speedup each algorithm accomplished relative to the fastest sequential time, and figure 5.2 depicts the corresponding efficiency curves, with efficiency measured in the conventional, $E = S/p$, way. In both graphs, dots show individual trials, and lines represent the mean value across the trials for each algorithm: the dotted line shows the Simple Distribution algorithm, the dashed line shows the results for the Multiple Distribution algorithm with queue size 6, and the solid line shows the mean speedup and efficiency for the Fault-Tolerant Distribution algorithm, also with queue size 6. Although this cluster has 32 processors available, due to practical considerations such the amount of time each run required, the necessity of sharing the cluster with other users, and the fact that regulations governing the machines' use typically restrict eight machines to interactive use only, the timing runs stopped at 24 rather than 32 nodes.

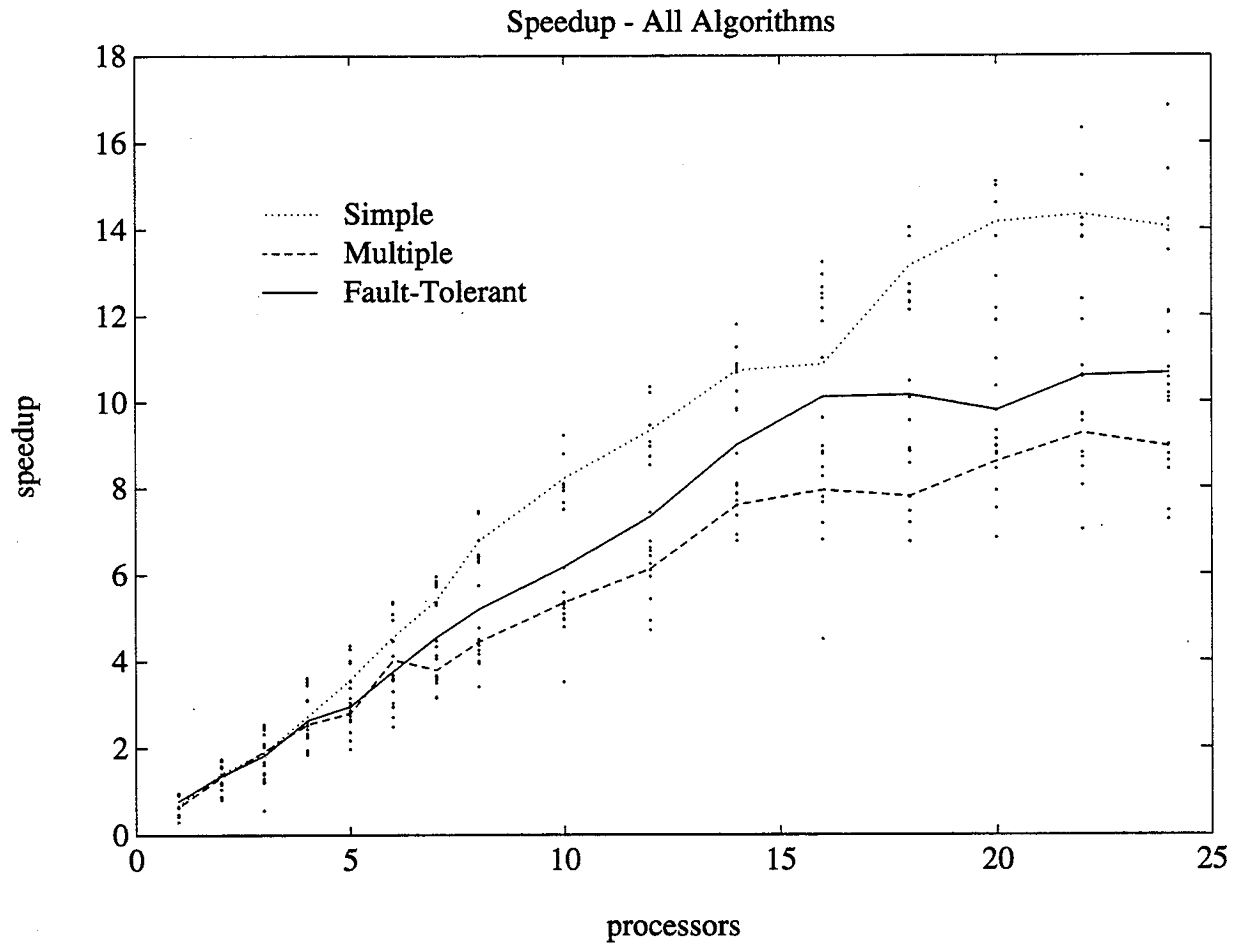

Figure 5.1: Mean speedup relative to fastest sequential time.

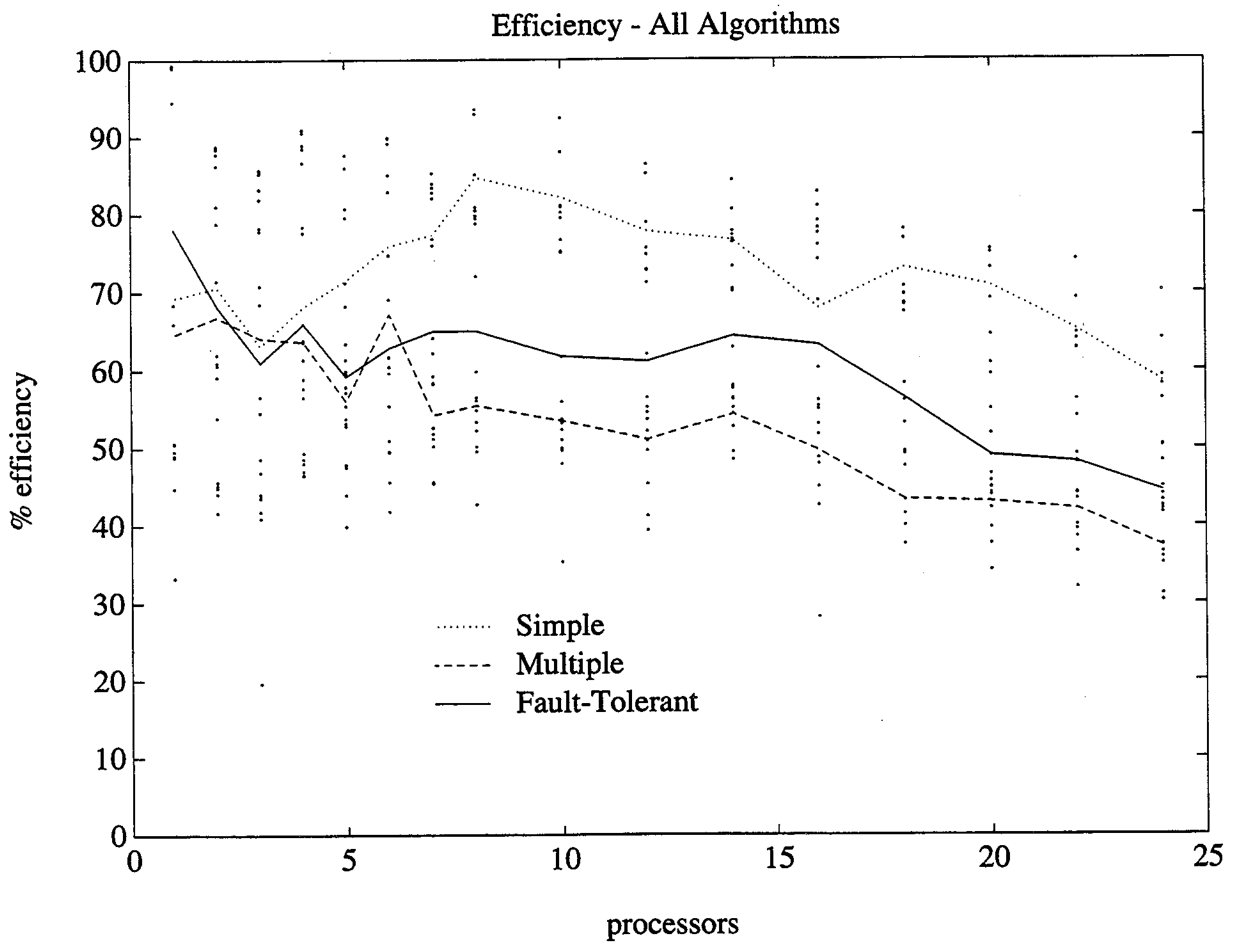

Figure 5.2: Mean conventional efficiency, $E = S/p$.

Perhaps the most interesting aspect of figure 5.1 is the unexpected result that both the Simple Distribution and the Fault-Tolerant Distribution algorithms outperformed the Multiple Distribution algorithm. Recall that the discussion in chapter 3 suggested that the Multiple Distribution algorithm should be more efficient than the Simple Distribution algorithm since it sends fewer messages. However, section 4.3 showed that in a heterogeneous system, if the individual processor speeds differ enough, the Simple Distribution algorithm should be faster than its Multiple Distribution counterpart. That situation may be occurring here: since several users shared the cluster during these timing runs, resulting in very heavy loads on some of the processing nodes, the processors in the experiment could have acted like the unbalanced system figure 4.1 depicts, even though all the node hardware was identical.

The performance of the Fault-Tolerant Distribution algorithm bears out this possibility: while it does not finish as quickly as the Simple Distribution algorithm because the nodes spend part of their time processing redundant information, it outperforms the Multiple Distribution algorithm. Both the Fault-Tolerant Distribution and the Multiple Distribution algorithms had the same queue size, and the model in Chapter 4 predicts that the Fault-Tolerant Distribution algorithm should behave similarly to the Multiple Distribution algorithm, but that it lacks the Multiple Distribution algorithm's minimum completion time constraint. Therefore, it appears that the minimum time constraint is coming into play: one or two slow nodes is seriously degrading the Multiple Distribution algorithm's performance.

Figure 5.2's efficiency curves track the speedup curves, as expected (recall that this figure shows conventional efficiency, $E = S/p$, which varies directly with S, the speedup.) Clearly, all three algorithms become less efficient as the number of processors increases. However, from the distribution of the dots, which represent individual trials, there appears to be a large random variation in the conventional measurement of efficiency for this multiuser system, as section 4.2 suggests there should be.

Note that none of the algorithms' speedup curves approaches a 1:1 slope (i.e. none of the algorithms approaches 100% conventional efficiency.) One reason for this lack of performance is the overhead Distributed NEPP introduces when running the NEPP code in parallel. A greater contributor to the problem, however, is the fact that there were several other users on the cluster during the timing runs, so Distributed NEPP rarely had access to the full computational power of the nodes. The sequential program, on the other hand, ran on an unloaded processor, so the curves in these figures would show a realistic picture of what engineers can expect when moving from their own, exclusive machines to a communally-shared cluster of processors.

Figure 5.3 shows the corrected efficiency for the three algorithms where, again, the solid line shows the Fault-Tolerant Distribution algorithm, the dashed line shows Multiple Distribution, and the dotted line shows Simple Distribution. Dots represent individual trials, as in figures 5.1 and 5.2. Recall that chapter 4 defines corrected efficiency as the amount of work the sequential program must perform to finish the task divided by the amount of computation the parallel version consumes. In this case, the sequential program requires an average of about 2155 CPU seconds on an IBM RS/6000 Model 560 to finish this particular input file; the CPU time actually varies somewhat from one run to the next, probably due to page swaps, network acquisition time, and other system overhead. The total CPU time for the parallel program is the sum of the CPU times for the host and all nodes. Dividing the sequential CPU time by the total parallel CPU time yields the corrected efficiency.

Figure 5.3 shows much less random variation than the conventional efficiency in figure 5.2, since corrected efficiency is less sensitive to varying load conditions. As the number of processors grows, both the Simple Distribution and Multiple Distribution algorithms' corrected efficiencies drop slightly from the additional overhead involved in coordinating the efforts of larger numbers of nodes. However, even at 24 nodes, these algorithms still have corrected efficiencies of about 85%, showing that for large numbers of processors, the Multiple Distribution and Simple Distribution algorithms consume about $1/0.85 = 117\%$ as much CPU time as the sequential version of the program with this input file. Also, figure 5.3 clearly shows the extra overhead the Fault-Tolerant Distribution algorithm introduces through processing redundant information. Its corrected efficiency starts at 100% at one node and drops to just over 60% at 24 nodes, at which point it consumes about 167% as much CPU time as its sequential counterpart given this particular input file.

Figures 5.4 through 5.6 provide a more detailed view of the random variation inherent in a multiuser system. These plots show the measured speedup for all three algorithms, with the solid line depicting the mean speedup and the dotted lines showing minimum and maximum measured values. Again, there were

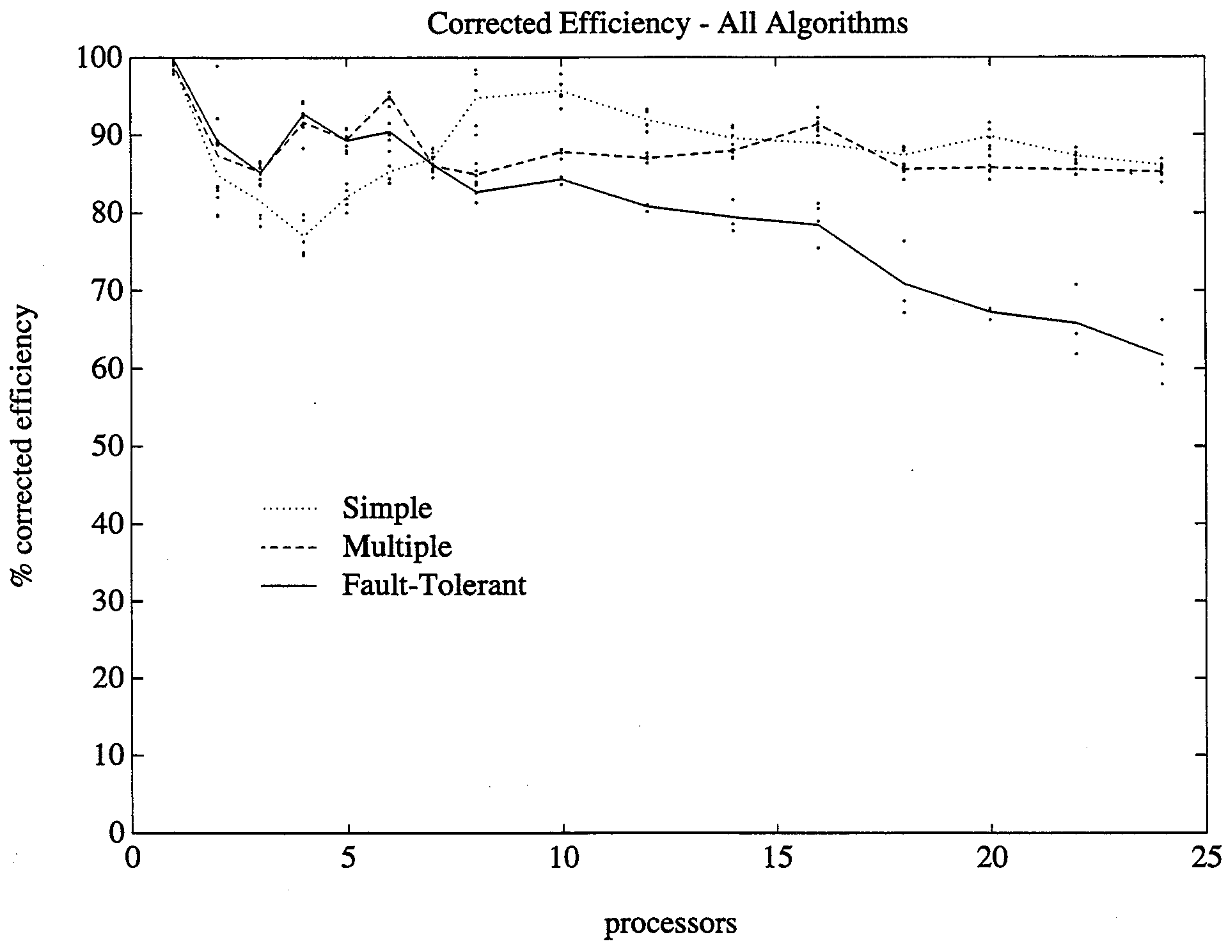

Figure 5.3: Mean corrected efficiency, $E = W_{req}/W_{par}$.

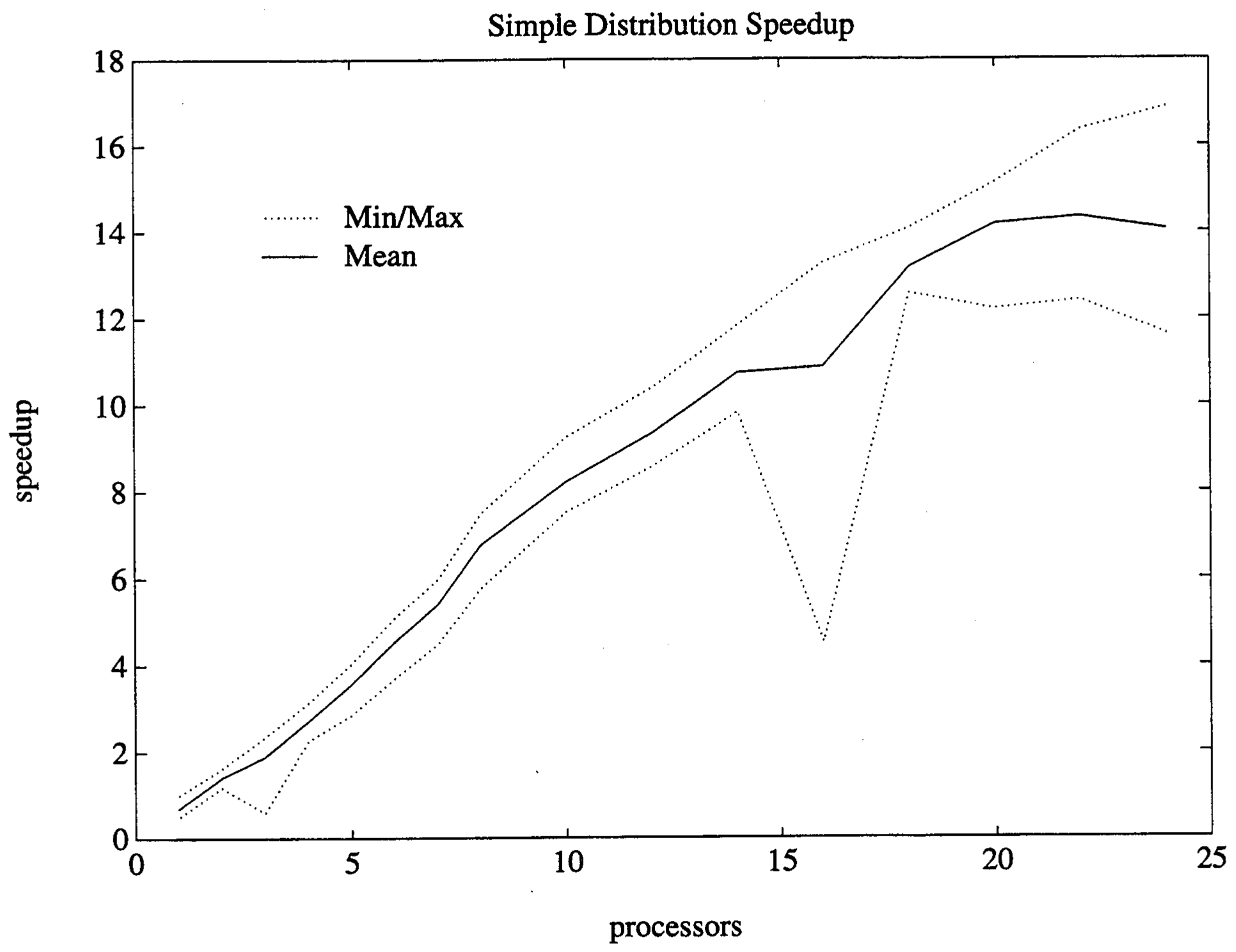

Figure 5.4: Simple Distribution algorithm speedup.

10 Fault-Tolerant Distribution trials and 5 trials each of the Multiple Distribution and Simple Distribution algorithms.

The features to note on these graphs are the spikes in the minimum speedup at 3 and 16 processors in figure 5.4 and in the maximum speedup at 6 processors in figure 5.5. Figure 5.6 shows similar spikes. At these points, abnormally high or low system load, or just an unusual load configuration, severely affected the measured speedup value. These random variations underscore the difficulty of taking reliable measurements in dynamic, multiuser systems and of applying those measured results to predicting the performance of any given program run.

5.3 Analysis of Model's Performance

The second set of Distributed NEPP experiments attempted to determine how accurately the analytical model in Chapter 4 agrees with observed data. The graphs in this section present their results in terms of percent relative error, where

$$\textit{percent relative error} = \frac{\textit{predicted} - \textit{observed}}{\textit{observed}} \times 100\% \qquad (5.1)$$

so that a relative error of +100% means that the model predicted a value twice as high as the observed value; similarly, a relative error of -100% means that the model underpredicted the observed value by a factor of

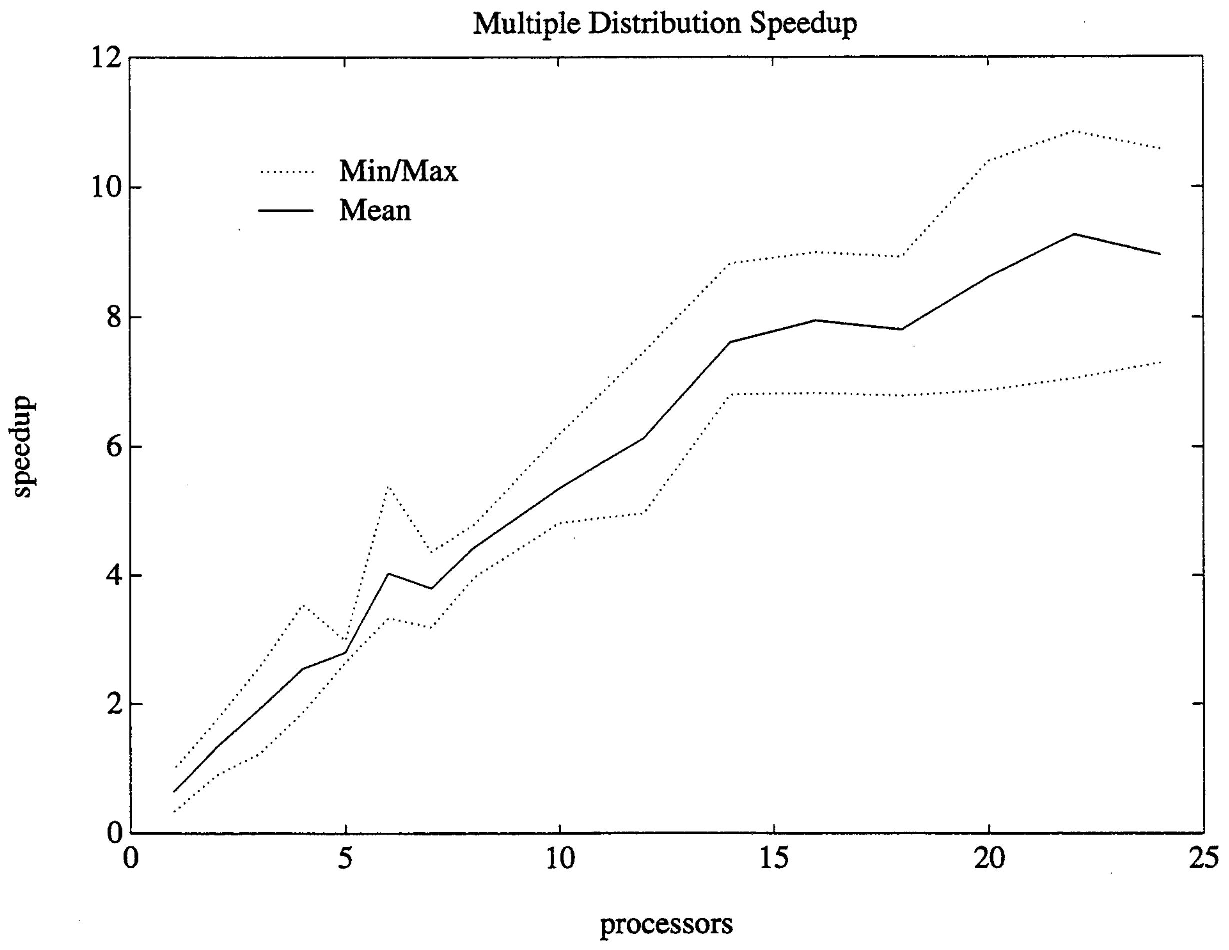

Figure 5.5: Multiple Distribution algorithm speedup.

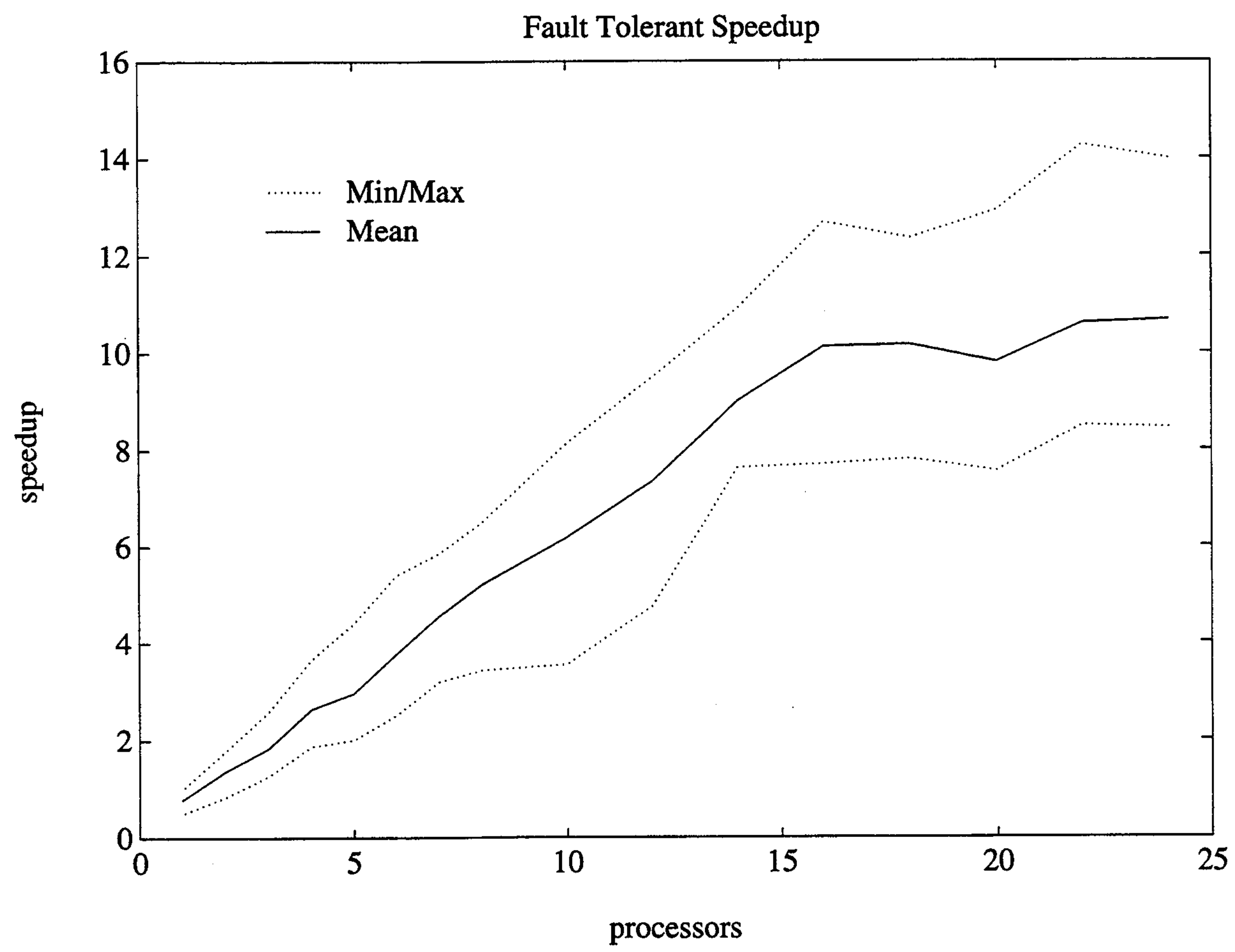

Figure 5.6: Fault-Tolerant Distribution algorithm speedup.

Table 5.1: Measured relative hardware speeds of different processor models.

model	**CPU time**	**rel. speed**
560	199.87	1.00
550	248.75	0.80
350	262.63	0.76
320	542.42	0.36

two.

The tests took place on a cluster of 12 IBM RS/6000's, including four model 550's, three model 560's, one model 350, and four model 320's. To determine the relative hardware speeds of the different types of machines, the researcher measured the aggregate CPU time each processor required to run the Fault-Tolerant Distribution version of Distributed NEPP with one node. Note that this trial used the parallel version of NEPP rather than the sequential version so the measure of relative hardware speed would include any variations in communication speed. Table 5.1 summarizes the relative hardware speeds of the different machines. This table normalizes the speed of the model 560 to 1.0, allowing the researcher to cross-check results with the machines in the cluster of model 560's section 5.2 discusses.

The next challenge was to measure w_i, the amount of work per job that the sequential and parallel versions of NEPP required. Since there is no clean way to determine how many standard instructions a program executes throughout its run, this measurement also used CPU time. In this case, the CPU time a given node consumed, multiplied by the node's relative speed from table 5.1, provides a measurement of the amount of computation the node used in "standard CPU seconds", the number of CPU seconds the computation would have required on an RS/6000 model 560. Note that converting from "standard CPU seconds" to "standardized instructions" requires simply multiplying the number of CPU seconds by a constant, the number of standard instructions the processor can execute in one second. Furthermore, a careful examination of the equations in chapter 4 reveals that if we measure relative speed (c_i) in terms of standard CPU time, and measure workload (w_j) the same way, then the total execution time equations return results in seconds, and the total work equations return results in terms of standard CPU time. Therefore, this analysis uses the relative speeds in table 5.1 as the values for c_i and uses CPU time to determine the values of w_j.

Finding f_j, the amount of overhead per job for the sequential program, is straightforward. In this case, running the sequential program on a model 560 showed that the program took 195.18 CPU seconds to finish an input file containing 300 jobs. Therefore f_j, or, more accurately, f_μ, the average value of f_j, is equal to $195.18/300 = 0.6506$ standard CPU seconds per job.

Measuring a_μ, the average amount of overhead each algorithm introduces per job, is more challenging. Recall from section 5.1 that message set-up time and time per byte differ significantly between communicating with a process on the same machine and communicating with one on a different machine. Furthermore, communication time represents a significant portion of the overhead each algorithm introduces. To arrive at an approximate number for the amount of overhead each algorithm introduces, the researcher measured the aggregate CPU time, in terms of standard CPU time, that Distributed NEPP required to run with 2 nodes under each algorithm. Dividing by the number of jobs yields an approximate value for work per job, and subtracting the value of f_μ above gives a value for a_μ. While this technique should be fairly accurate for the Simple Distribution and Multiple Distribution algorithms, it cannot help but include the time for a few redundant job results under the Fault-Tolerant Distribution algorithm. The researcher expected that the large number of jobs, along with the fact that with only two processors the Fault-Tolerant Distribution algorithm tends to generate few redundant jobs, would tend to swamp the measurement error.

The final constants the model needs are α_h and α_i, the fraction of time Distributed NEPP has access to the CPU of the host and each node processor during its run. Distributed NEPP gathers and returns the amount of CPU time it consumes on each node processor, in terms of the node processor's own CPU time rather than standard CPU time. Dividing each node's CPU time by the total application run time provides

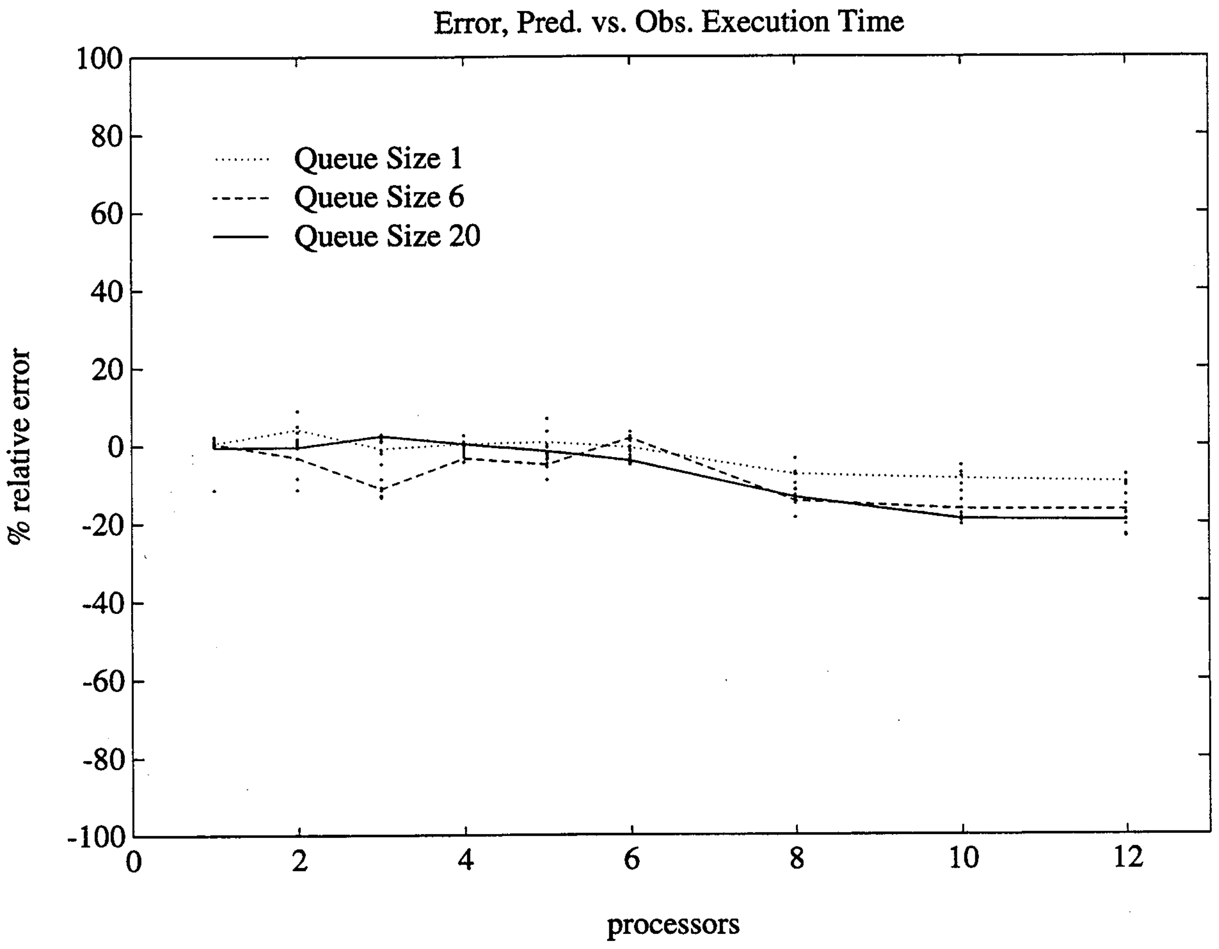

Figure 5.7: Relative error in Multiple Distribution algorithm, predicted versus observed wall clock time.

a value for each of the α_i and for α_h. (As an example, if Distributed NEPP consumed 30 CPU seconds on a node during an application run that took 40 seconds wall clock time, then it spent 30 seconds on that node's CPU during the application run, so that node's $\alpha_i = 30/40 = 0.75$.)

Figure 5.7 shows the relative error between predicted and observed wall clock time for the Multiple Distribution algorithm. Figure 5.8 shows the error in standard CPU time for the same algorithm. Dots represent individual trials, and lines show mean values: the dotted, dashed, and solid lines correspond to queue sizes 1, 6, and 20, respectively. (Recall that the Simple Distribution algorithm is equivalent to the Multiple Distribution algorithm with queue size 1.) The agreement between the model and experimental data tends to be fairly good in general, though the model becomes slightly optimistic as the number of nodes increases.

Figures 5.9 and 5.10 show the same data for the Fault-Tolerant Distribution algorithm, where figure 5.9 is the error in wall clock time and 5.10 is the error in CPU time. Again, dots represent individual trials and the dotted, dashed, and solid lines show mean values for queue sizes 1, 6, and 20, respectively. Here the model tends to overpredict the observed data, with the degree of error growing with queue size.

One possible reason for the model's increasing pessimism is a simplifying assumption that equation 4.13 makes. Here is the equation again, reproduced for convenience:

$$\frac{dx}{dt} = -\left(\frac{x}{pq}\right)\left(\frac{ps_\mu}{w_\mu}\right) \tag{5.2}$$

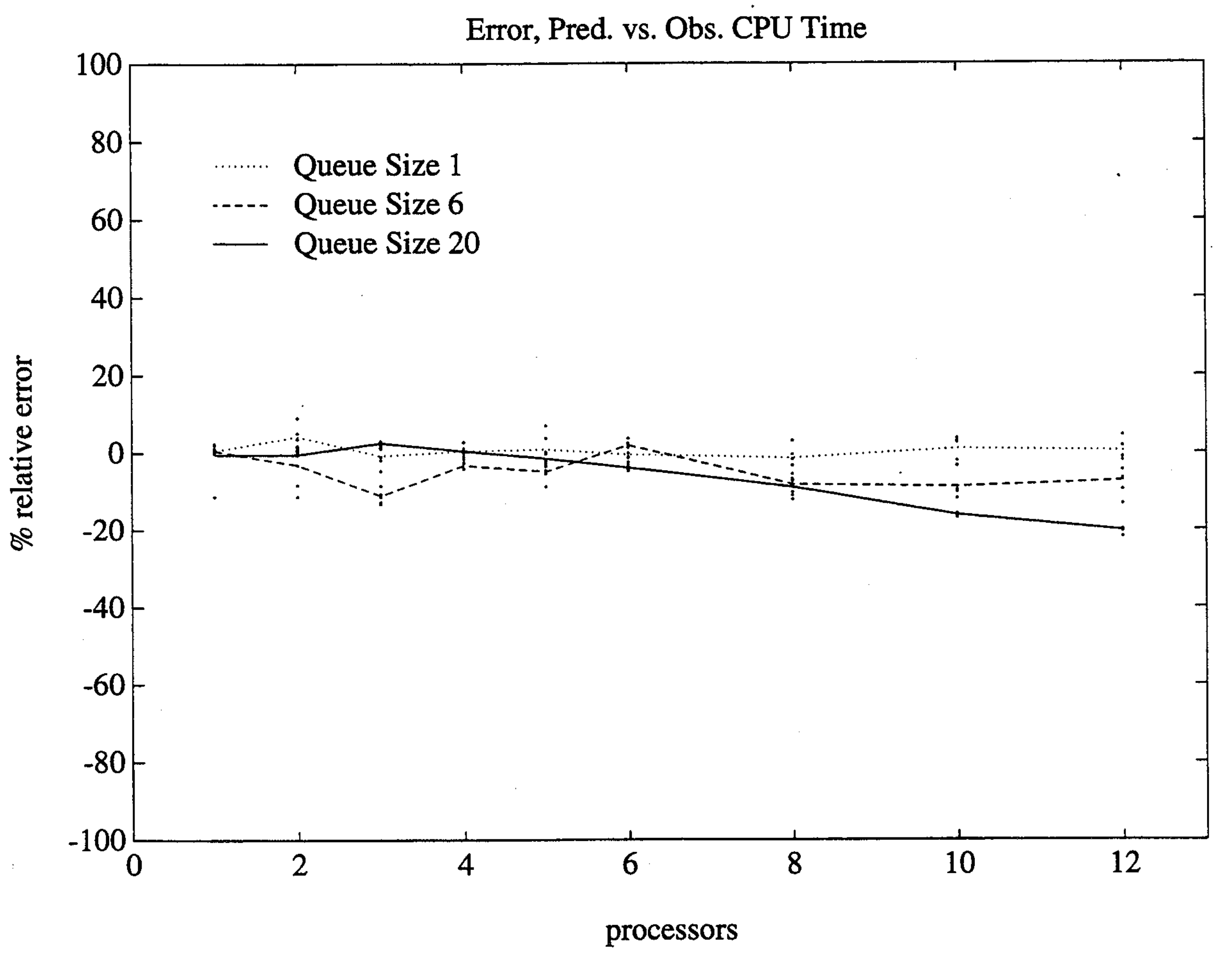

Figure 5.8: Relative error in Multiple Distribution algorithm, predicted versus observed standard CPU time.

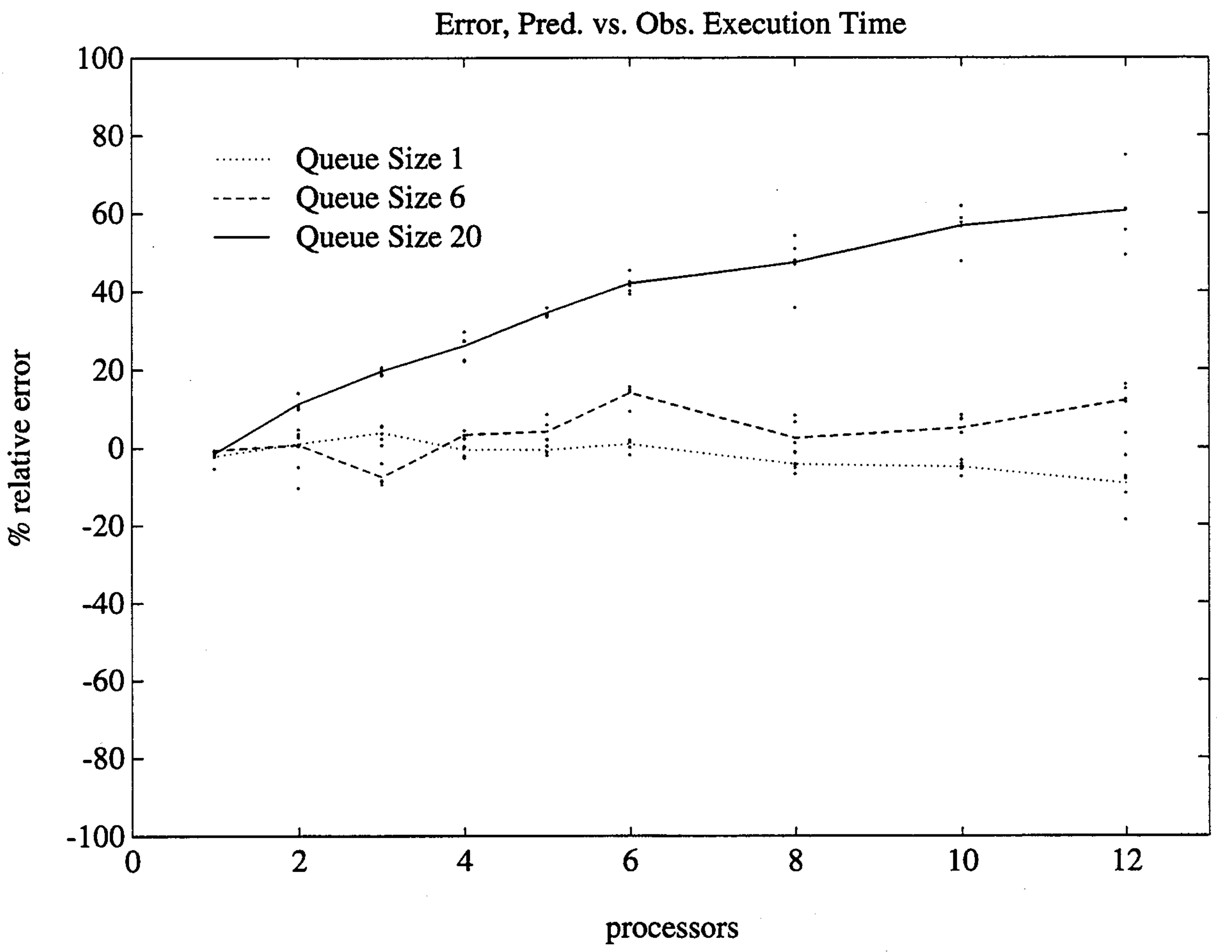

Figure 5.9: Relative error in Fault-Tolerant Distribution algorithm, predicted versus observed wall clock time.

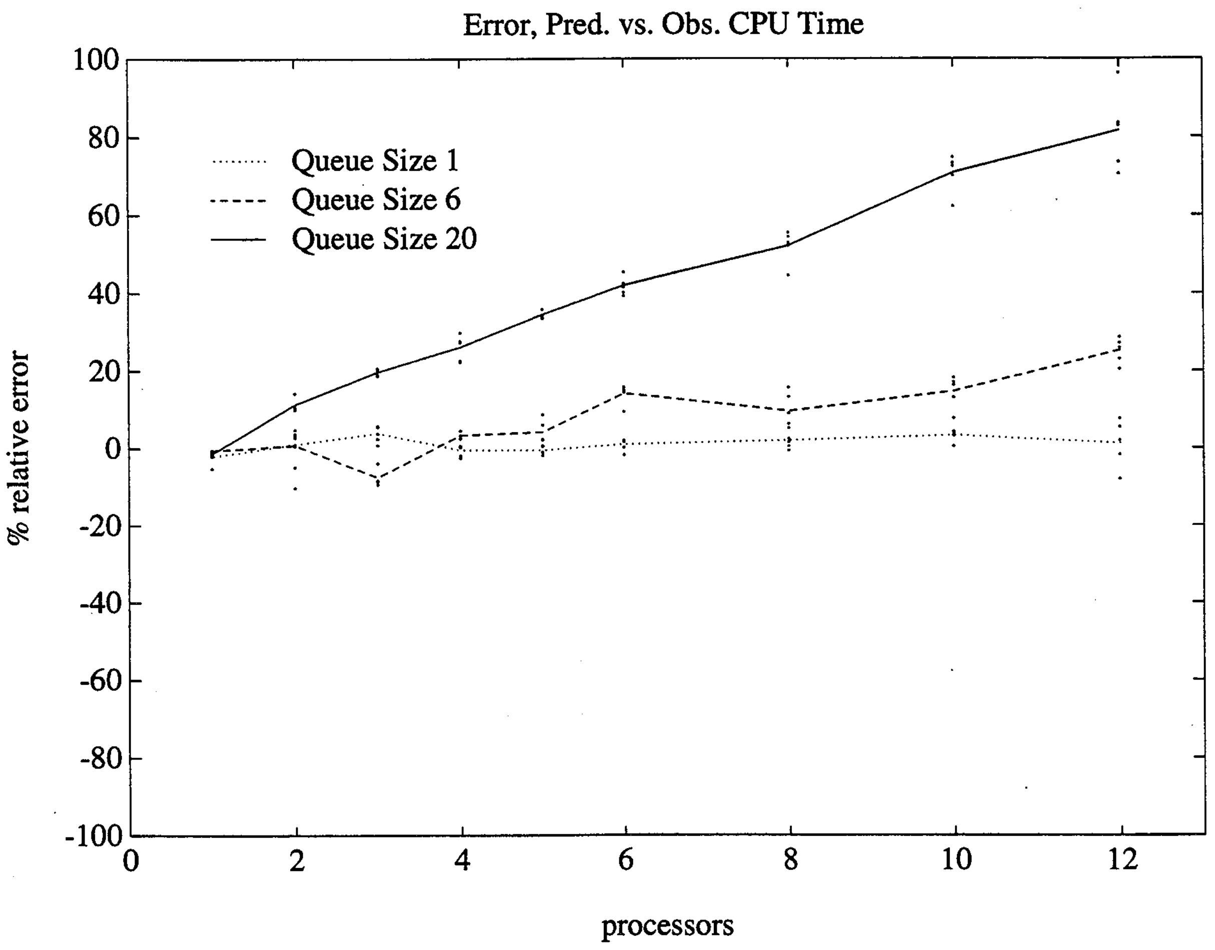

Figure 5.10: Relative error in Fault-Tolerant Distribution algorithm, predicted versus observed standard CPU time.

In this equation, the ps_μ/w_μ term represents the number of job results the nodes return to the host per unit time, and x/pq represents the fraction of those results that are not redundant. In essence, this equation assumes that redundant and non-redundant results from the nodes have a uniform distribution. However, under the Fault-Tolerant Distribution algorithm, it is the first copy of any job's results that the host receives that is non-redundant: all succeeding copies of that job's results are redundant. Furthermore, once the host receives that first copy of the results, it stops making copies of that job, reducing the overall number of redundant jobs. Since the error grows with queue size, suggesting that the problem may lie with overpredicting the number of redundant results, incorporating a more accurate model of the queues and distribution of redundant and non-redundant results would probably reduce the model's mean relative error.

Chapter 6

Additional Research and Conclusions

6.1 Additional Research Possibilities

This research has brought forth a number of interesting possibilities that deserve exploration. Here are some of them.

6.1.1 Dequeuing Redundant Jobs

One of the factors that hurts the performance of the Fault-Tolerant Distribution algorithm is the fact that the nodes process a large number of redundant jobs. One possibility that might increase the algorithm's performance significantly is to have the host periodically compile a list of all jobs that are now redundant and broadcast that list to the nodes. When the nodes receive the list, they check it against the jobs in their job queues and dequeue any redundant jobs. The host would also have to dequeue the corresponding jobs from its internal copies of the nodes' job queues so that it could tell how many jobs each node had it its queue. Unfortunately, the relatively simple model section 4.1 presents is not sophisticated enough to predict the Fault-Tolerant Distribution algorithm's performance if the host where to dequeue redundant jobs.

6.1.2 Distributing the Host's Duties

In Distributed NEPP, the host is both a bottleneck and a vulnerable point in the system: if the host fails, the whole application will fail with it. Furthermore, the host's workload grows with the number of nodes; eventually, the host will reach a point where it cannot handle any more nodes, at which point the application's speedup curve will level off. It would be useful to distribute the host's tasks to improve fault-tolerance and to avoid the bottleneck problem.

Any version of Distributed NEPP with a distributed host process still has to operate under the constraint that only one machine in the cluster has access to the input and output files. Therefore, some portion of the application will have to be centralized. However, it should be possible to minimize the application's exposure to crashing due to a single-point failure by copying and distributing the input files and keeping local copies of results so other programs could recover the output file after a critical failure.

The simplest way to modify Distributed NEPP would be to have multiple hosts, each with its own, independent compliment of nodes. A *fileserver* process, located on the only machine with access to the files, is responsible for copying and distributing the input files and writing results to the output files (figure 6.1.)

The user starts the fileserver process. The fileserver, in turn, starts the host processes and multicasts a message to the hosts containing a complete copy of the input files and the number of hosts in the system. Giving each host a copy of the input files ameliorates the limitation that only 1 machine can access the files.

Once the hosts have the input files from the fileserver, each chooses a range of input jobs to process based on its own host number. For instance, if there are a total of J jobs to process, host i might choose the job range starting at job number $i\lfloor J/h \rfloor$ and extending for the next $\lfloor J/h \rfloor$ jobs, where h is the number of hosts. Each host also creates a table of all possible job ranges and marks each range as "NOT FINISHED".

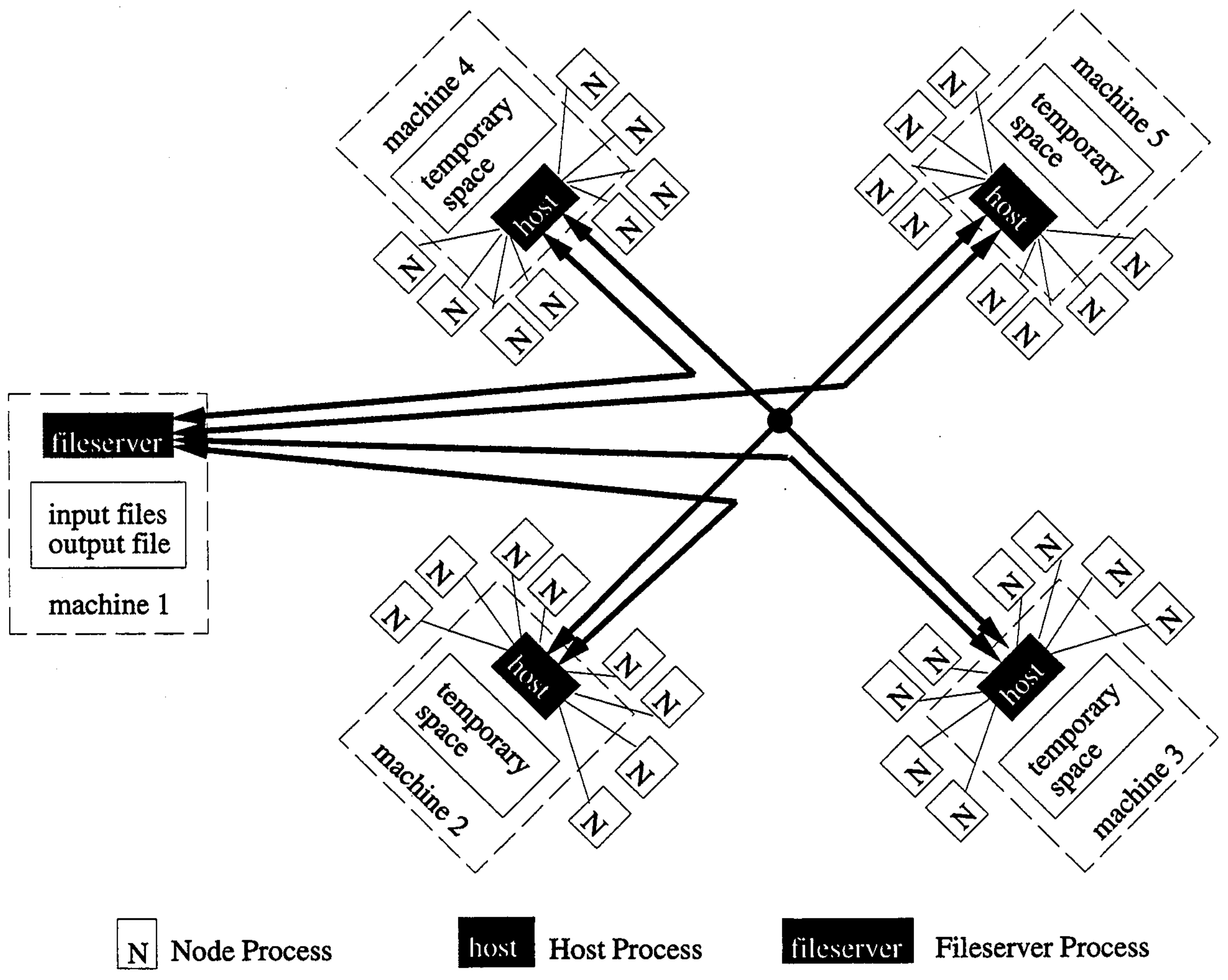

Figure 6.1: Multiple host modification to Distributed NEPP.

The hosts start node processes on other machines and calculate the results to the jobs in their chosen ranges just as in the current version of Distributed NEPP. When a host finishes its range of jobs, it writes all the results for that job range to the temporary file space on the machine on which it is running, sends a copy of the results to the fileserver, multicasts a message to the other hosts telling them that it has finished with its job range, and chooses a new range of jobs to work on. Upon receiving the message telling them a host has finished a particular range of jobs, all hosts, including the one that sent the message, mark that particular range of jobs "FINISHED" in their job range tables.

When the fileserver receives results from a host, it puts those results into a sorted list of results to write to the output file. If it already has a copy of the results, it simply discards them. Only after it has written a host's results to the output file does the fileserver send the host a message confirming that fact. Once the host receives confirmation from the fileserver, it is free to delete the results file for that particular range from its local machine's temporary file space. This part of the protocol ensures that if the fileserver process should crash, the hosts will accumulate results locally. Then, another program could go through the host machines, collect the temporary files, and rebuild the output file the fileserver was trying to create.

Whenever a host has finished a range of jobs, or if it receives a message indicating that a different host has finished the jobs it was working on, the host has to choose a new range of jobs to process. It repeatedly applies a table-walk function to the current job range to search the job range table for a range marked "NOT FINISHED". When it finds such a range, it begins processing those jobs. If a host searches the whole job range table and finds only "FINISHED" jobs, it knows that all the jobs in the files it possesses are done, leaving it free to send a "DIE" message to its nodes and exit. The table-walk function the host uses could be as simple as taking the current job range and choosing the next higher one, or it could be modeled after a sophisticated hash function designed to distribute the workload randomly and avoid having two hosts arbitrarily choose the same job range. The only requirement for this function is that repeatedly applying it will result in its visiting every range in the job range table once before eventually returning to the original starting range [3].

By having only the hosts and the fileserver send messages back and forth, this scheme allows the communications traffic each host must handle to grow slowly with the number of nodes: if each host has n nodes, then the host must communicate with its own n nodes, receive broadcasts from other hosts, and communicate with the fileserver. If the total number of nodes in the system is N, then the number of processes with which a given host must communicate will be $n + \log_n N$.

On the other hand, this approach is still vulnerable to host failure in the sense that each failed host removes n node processors from the pool available to the application as a whole. One solution to the problem might be to have each host periodically check in with a neighbor, so that if a host fails to check in within a specified time frame, another host can signal the fileserver to kill the host that failed to check in, if necessary, and to start a new host to replace it.

6.1.3 Recovering From a Node Crash

Under the Fault-Tolerant Distribution algorithm, if the host wishes to restart a failed node, it has to make a system call to create the node process, send that process any initialization information it needs—the NEPP design point in this case—and create an empty internal queue to mirror the new node's job queue. As soon as the new node process signals the host with a "READY" message, the host will begin sending it work just as it would with any other node.

Unfortunately, there does not appear to be a clean way for the host to detect the fact that a node has crashed. One of the assumptions that underlie the Fault-Tolerant Distribution algorithm is that a node could be running arbitrarily slowly, so it is difficult for the host to tell the difference between a very slow node and one that has died. PVM includes library calls that allow one process to check to see if another is still alive, but even having such library calls available still leaves the fundamental problem of when the host should check. In other words, when does the host have enough evidence available to believe that a node is dead, or at least to check on a particular node?

It may be that, with the Fault-Tolerant Distribution algorithm, the most practical approach would be to have the host check for dead nodes with a frequency that depends on how long it takes to finish the particular application and the amount of time necessary to check for and restart crashed nodes. For instance, if it takes much longer to restart a node than it does to finish all the jobs in the input file, then the application will run more quickly if it just decreases the number of processing nodes by 1 and continues running the application, which the Fault-Tolerant Distribution algorithm does automatically. On the other hand, if the time to check for and restart a dead node is much shorter than the time it would take to finish the application, then recovering a dead node could have a significant impact on the total run time, and it will be worth the cost of restarting it. Also, if the host has a relatively light workload, it may be able to check for dead nodes while still efficiently coordinating the efforts of functional processing nodes, in which case it could check more often.

6.2 Conclusions

Distributed NEPP meets the design goals for converting NEPP to run in parallel: the new program shows parallel speedup in a distributed environment consisting of a set of network-connected workstations. While others have proposed mechanisms that will allow an application to run in parallel in a distributed environment [2] [15], improve the reliability of a distributed application [12], or both [13], the particular constraints of this project suggest that implementing these approaches would be inefficient or overly complex. Instead, Distributed NEPP uses a centralized control scheme that provides efficiency and fault-tolerance, and which better fits the program's design constraints.

Distributed NEPP's design evolved through three successive algorithms: Simple Distribution, Multiple Distribution, and Fault-Tolerant Distribution. A closer inspection shows that the Simple Distribution algorithm is a special case of the Multiple Distribution algorithm. Furthermore, the Fault-Tolerant Distribution algorithm incorporates the Multiple Distribution algorithm but adds redundancy in time leading to lower wall-clock times in systems with large load variations and the ability to tolerate node failures, at the expense of greater system-wide CPU usage.

Experiments show that the Multiple Distribution algorithm can run slower than the Simple Distribution algorithm under certain load conditions. Analytical modeling bears out the observation: the Multiple Distribution algorithm is much more sensitive to variations in processor speed across the system. In fact, the model predicts that the minimum execution time for the Multiple Distribution algorithm varies directly with the nodes' queue sizes, so large queues, which should speed up the algorithm by making it more efficient, can have the opposite effect in a heterogeneous system. The Fault-Tolerant Distribution algorithm does not suffer from the same minimum time constraints, although the redundancy it incorporates tends to make it slower than the Simple Distribution algorithm unless the system load or processor speeds are extremely unbalanced or a node fails.

Experimental tests of the analytical model indicate it has a tendency to slightly underpredict the CPU usage and wall clock times of the Simple Distribution and Multiple Distribution algorithms; the model's error grows with increasing numbers of processors. Furthermore, the model overpredicts the CPU and wall clock times for the Fault-Tolerant Distribution algorithm. In this case, the model's error seems to arise from simplifying assumptions it makes about the random distribution of redundant and non-redundant information in the results the nodes send to the host, leading to an error that increases with both number of processors and queue size.

Finally, a number of possibilities for enhancing Distributed NEPP and its associated algorithms arose through the course of this research. Modifying the Fault-Tolerant Distribution algorithm to dequeue results the host knows are redundant should improve the algorithm's performance while decreasing its impact on system load. Also, giving Distributed NEPP the ability to recover and reintegrate faulty nodes rather than simply rerouting work to other nodes would be a straightforward modification under the Fault-Tolerant Distribution algorithm. Finally, it should be possible in the future to distribute the program's centralized control functions, decreasing the potential for a single-point failure and to increasing the number of processors Distributed NEPP can use before the control system saturates.

Appendix A

How To Use Distributed NEPP

Distributed NEPP uses the same format for its input files that the sequential version of NEPP uses. Unfortunately, that format is too long and complex to reproduce here: please see [9] for a complete description of the input file format.

Instead of prompting the user for the names of the various input files it needs, Distributed NEPP expects to read the file names from a *defaults file*, a file of file names. The defaults file should have five lines, with one file name per line, in this order:

1. input file name
2. map file name
3. thermal map file name
4. output file name
5. *Instal* code file name

If the engine the input file specifies does not use NEPP's thermal equilibrium package, leaving line 3 blank will shorten the program's execution time by a few seconds because the host program will not attempt to copy the file and send it to the nodes. (Please note, however, that the defaults file must still have five lines—leave line 3 blank rather than deleting it altogether if the engine does not use the thermal code.)

Another optimization Distributed NEPP incorporates is that instead of immediately copying all input files and sending them to the nodes, the host program first sends the nodes the names of the input files. The nodes then try to open the input files themselves, an attempt that will work if both the host and nodes are running under a network file system. If the nodes cannot open the input files, they send a message back to the host requesting the files' contents. The host then opens the files, copies their contents, and sends the contents to the nodes. The practical implication of this process is that putting a complete path name, rather than just a relative path name, on each line of the defaults file may improve Distributed NEPP's performance by a few seconds. Figure A.1 shows a sample defaults file.

```
~/dnepp/input/tbe.input
~/dnepp/maps/tbe.maps
/usr/lib/nepp/small.btherm
~/dnepp/output/tbe.output
~/dnepp/maps/instal.maps
```

Figure A.1: Example defaults file for Distributed NEPP.

```
# comment lines begin with a "#" character
machine1.somewhere.someuniversity.edu
machine2.somewhere.someuniversity.edu
machine3.somewhere.someuniversity.edu
```

Figure A.2: Example PVM hosts file.

The remaining file Distributed NEPP needs to run under PVM version 2.4 is a *hosts* file for PVM. The hosts file tells PVM on which machines it may place Distributed NEPP's nodes. In its simplest form, the hosts file is a list of machine names, with one name per line of the file. Please see [5] for a more complete description of the hosts file format. Figure A.2 shows a sample PVM hosts file.

Running Distributed NEPP requires only three steps: start PVM, run the host program, and, when the program has finished processing, stop PVM. To start PVM version 2.4, type "pvmd myhosts &", where "myhosts" is the PVM hosts file, and the ampersand character, "&", directs Unix to put the PVM process in the background, leaving the terminal free so the user can run other programs. Once the master PVM daemon has started slave daemons on all the machines in the hosts file, it signals that fact by printing the message "pvm is ready".

Once PVM is ready, the next step is to run the Distributed NEPP host program. The name of the host program could vary from site to site, but, for the sake of this discussion, assume it is called "nepphost". To run it, type "nepphost defaults.def", where "defaults.def" is the name of the Distributed NEPP defaults file. "nepphost" automatically finds out how many machines are available to it, starts a copy of the node program "neppnode" running on each available machine, and reads and processes the input file.

It is possible to run "nepphost" several times without restarting PVM. However, after the last Distributed NEPP run, remove PVM by first typing "fg" to bring the PVM process to the foreground, so it can accept commands from the keyboard, and then typing control-C ("^C") to kill the process.

Appendix B

The Distributed NEPP Source Code

Distributed NEPP is a complex application consisting of many interacting modules. Table B.1 provides a summary of the Distributed NEPP source files and what they contain. The rest of this appendix provides more detail on the files' contents.

Distributed NEPP uses a fairly sophisticated *make* file to give it the ability to compile and run on the Sun SPARC and the IBM RS/6000, with the potential for adding more architectures in the future. The Make file runs the shell script "set_system" to determine the system architecture and to set environment variables so the code will compile properly; anyone installing Distributed NEPP on a new system should edit "set_system" to reflect the correct system architecture. Also, the person installing Distributed NEPP should edit all the path names in the Make file so it can find the various libraries and source files Distributed NEPP requires.

The Distributed NEPP source code requires two libraries in addition to the standard C and Fortran library files. The first is the library of PVM system calls [5]. Distributed NEPP also requires a library containing NEPP Fortran routines [10], which it calls to perform the actual analysis. Please refer to the documentation that comes with the PVM and NEPP packages for instructions on creating these libraries.

The file "nepphost.c" contains the majority of the code for Distributed NEPP's host program. The host is event-driven: it generates an event number from the current system state and a list of event priorities. Then it takes an action based on the even number. This event-driven approach allows the host to schedule a complex sequence of actions. Furthermore, it lets different interfaces set different priorities; in particular, "neppint.c" and "ipasint.c" each contain their own, individual routines to generate event numbers.

In contrast, the node program is relatively simple. "neppnode.c" contains most of the node's code. The nodes do not use an event-driven design because they tend to behave in the same way regardless of what the host does. Therefore, their code focuses more on simplicity than on flexibility. "neppnode.c" also has a preprocessor symbol that determines, at compile time, the node's priority level under Unix; the nodes do not give the end user the option of changing their priority levels at run time because, in an office-type environment, there is a good chance that engineers will not want other users' node programs on their machines if they know the other users could run the nodes at high priority and thereby slow down the engineer's own programs.

"neppint.c" and "ipasint.c" provide front-end interfaces for Distributed NEPP. "neppint.c" creates the stand-alone version of Distributed NEPP that Appendix A describes. "ipasint.c", on the other hand, creates a version of Distributed NEPP that other programs, like IPAS [8], can use to perform engine analysis in parallel.

The header files "parnepp.h" and "host.h" contain a number of switches to configure Distributed NEPP's behavior. By changing preprocessor symbols in "host.h", the person compiling the code can determine whether or not Distributed NEPP will print a message whenever it receives job results from a node. During development, and for input files requiring lots of computation, providing the end user with feedback about Distributed NEPP's progress allows the user to tell Distributed NEPP is still running and has not crashed.

"parnepp.h" includes several useful switches. It allows the person compiling the code to turn on Distributed NEPP's diagnostic printouts and internal error checking routines. It also includes a switch that will

Table B.1: Summary of Distributed NEPP source files.

Makefile	file to direct Unix *Make* facility to compile and assemble Distributed NEPP
set_system	C shell script to set architecture-dependent environment variables
commo.h	data structures for communications module
files.h	data structures for file handling routines
host.h	data structures and declarations specific to host program
linkfix.h	type definitions and declarations to interface between C and Fortran
parnepp.h	general configuration and data structure header file
pvmuser.h	PVM header file, comes with PVM distribution package
commo.c	communications module
dneppmain.c	the C "main" function
files.c	module allowing C code to manipulate Fortran files
ipasint.c	interface between NEPP and IPAS
linkfix.c	interface between C function calls and Fortran procedures
nepphost.c	majority of the code for the Distributed NEPP host program
neppint.c	user interface for stand-alone Distributed NEPP (as opposed to ipasint.c: interface between NEPP and IPAS)
neppnode.c	majority of the code for the Distributed NEPP node program
filehandler.f	Fortran side of the interface that allows C routines to manipulate Fortran files
avceng.f	routines to interface to NEPP Fortran library to perform engine analysis
tread.f	routines allowing Fortran code to read the NEPP input files

cause Distributed NEPP to gather statistics on the wall clock and CPU time it consumes during each run, simplifying the process of profiling the application's performance. It defines C data structures that mimic the Fortran common blocks the NEPP library uses, so that Distributed NEPP can access common block variables. Finally, this header file controls the nodes' queue sizes, the threshold point at which the host regards a node as being idle, the number of jobs the host packs into each assignment message, and whether or not the host can send replicas of jobs to idle nodes: by editing this file, a developer can control whether Distributed NEPP uses the Simple Distribution, Multiple Distribution, or Fault-Tolerant Distribution algorithm.

The file "commo.c" contains the communications routines that interface between Distributed NEPP and PVM. Both the host and the nodes access the routines in this file; putting both host and node communications routines in the same file helps to reduces programming errors in which the sender and the receiver expect slightly different message formats. The header file "commo.h" provides access to the functions in this module.

The two files "linkfix.h" and "linkfix.c" provide the rest of Distributed NEPP with a system-independent interface between C and Fortran. Developers who are porting Distributed NEPP to a new system or set of compilers will probably have to modify these files to handle the new compilers' protocols for calling Fortran subroutines and accessing Fortran variables from C.

"files.h", "files.c", and "filehandler.f" provide a set of routines that allow C functions to manipulate Fortran's file units. Other Distributed NEPP routines use these functions to open and close NEPP input and output files and to create temporary files that the Fortran subroutines in the NEPP library will use.

Finally, "avceng.f" and "tread.f" act as hooks into the NEPP library. Distributed NEPP calls the subroutines in these modules to read design and off-design cases from the NEPP input files and to process engine data.

Bibliography

[1] Bannister, Joseph A., and Kishor S. Trivedi. "Task Allocation in Fault-Tolerant Distributed Systems." *Acta Informatica.* v 20 (1983): 261-81.

[2] Blake, Ben A. "Assignment of Independent Tasks to Minimize Completion Time." *Software—Practice and Experience.* v 22 n 9 (September 1992): 723-34.

[3] Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* U.S.A.: MIT Press and McGraw-Hill Book Company, 1990. 219-43.

[4] Cours, Jeffrey, and Brian Curlett. *A Distributed Version of the NASA Engine Performance Program.* NASA TM-106208. Cleveland: NASA Lewis Research Center, 1993.

[5] Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *Parallel Virtual Machine.* Version 2.4. Computer software. Available through Internet FTP from public archive sites, 1992.

[6] —. *Parallel Virtual Machine.* Version 3.0. Computer software. Available through Internet FTP from public archive sites, 1993. Beta test version.

[7] Grošelj, Bojan. "Fault-Tolerant Distributed Simulation." *1991 Winter Simulation Conference Proceedings.* Piscataway: IEEE Service Center (December 1991): 637-41.

[8] Lavelle, T. M., R. M. Plencner, and J. A. Seidel. *Concurrent Optimization of Airframe and Engine Design Parameters.* NASA TM-105908. Cleveland: NASA Lewis Research Center, 1992.

[9] Plencner, R. M., and C. A. Snyder. *The Navy/NASA Engine Program (NNEP89)—A User's Manual.* NASA TM-105186. Cleveland: NASA Lewis Research Center, 1991.

[10] NEPP distribution: for availability contact Brian P. Curlett, Aeropropulsion Analysis Office, NASA Lewis Research Center, Cleveland, OH, 44135. U.S. Phone (216) 977-7041. Internet E-mail "curlett@hornet.lerc.nasa.gov".

[11] Sahni, Sartaj, and Yookun Cho. "Scheduling Independent Tasks with Due Times on a Uniform Processor System." *Journal of the Association for Computing Machinery.* v 27 n 3 (July 1980): 551-63.

[12] Shatz, Sol M., Jia-Ping Wang, and Masanori Goto. "Task Allocation for Maximizing Reliability of Distributed Computer Systems." *IEEE Transactions on Computers.* v 41 n 9 (September, 1992): 1156-68.

[13] Smith, Reid G. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver." *IEEE Transactions on Computers.* v C-29 n 12 (December 1980): 1104-13.

[14] Stone, Harold S. *High Performance Computer Architecture, 3rd ed.* Reading: Addison-Wesley, 1993. 158-62.

[15] Vaughan, J. G. "Static Performance of a Divide-and-Conquer Information-Distribution Protocol Supporting a Load-Balancing Scheme." *IEE Proceedings, Part E: Computers and Digital Techniques.* v 139 n 6 (September 1992): 430-38.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 1994	**3. REPORT TYPE AND DATES COVERED** Final Contractor Report

4. TITLE AND SUBTITLE Design and Implementation of a Distributed Version of the NASA Engine Performance Program	**5. FUNDING NUMBERS** WU-505-69-50 C-NAG3-1369
6. AUTHOR(S) Jeffrey T. Cours	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ohio State University, Department of Electrical Engineering, 2015 Neil Avenue, Columbus, Ohio 43210	**8. PERFORMING ORGANIZATION REPORT NUMBER** E-8619
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration, Lewis Research Center, Cleveland, Ohio 44135-3191	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** NASA CR-194475

11. SUPPLEMENTARY NOTES

Project Manager, Brian P. Curlett, Aeropropulsion Analysis Office, organization code 2410, NASA Lewis Research Center, (216) 977-7041.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 61	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 words)***

Distributed NEPP is a new version of the NASA Engine Performance Program that runs in parallel on a collection of Unix workstations connected through a network. The program is fault-tolerant, efficient, and shows significant speedup in a multi-user, heterogeneous environment. This report describes the issues involved in designing Distributed NEPP, the algorithms the program uses, and the performance Distributed NEPP achieves. It develops an analytical model to predict and measure the performance of the Simple Distribution, Multiple Distribution, and Fault-Tolerant Distribution algorithms that Distributed NEPP incorporates. Finally, the appendices explain how to use Distributed NEPP and document the organization of the program's source code.

14. SUBJECT TERMS Distributed processing; Parallel programming; Cycle analysis; Engine performance	**15. NUMBER OF PAGES** 59
	16. PRICE CODE A04

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT**